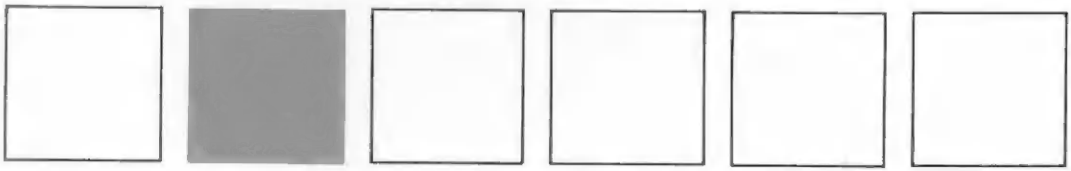
HEWLETT PACKARD

HP 82480A

Math Pac

Owner's Manual

For the HP-71



Notice

Hewlett-Packard Company makes no express or implied warranty with regard to the keystroke procedures and program material offered or their merchantability or their fitness for any particular purpose. The keystroke procedures and program material are made available solely on an "as is" basis, and the entire risk as to their quality and performance is with the user. Should the keystroke procedures or program material prove defective, the user (and not Hewlett-Packard Company nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages. Hewlett-Packard Company shall not be liable for any incidental or consequential damages in connection with or arising out of the furnishing, use, or performance of the keystroke procedures or program material.

Math Pac

Owner's Manual

For Use With the HP-71

March 1984

82480-90001

Printed in Singapore

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Edition 1 ..... Mar 1984; Manufacturing Part Number 82480-90011
Manual Update ..... Aug 1984; Manufacturing Part Number 82480-90015
Edition 2 ..... Sep 1984; Manufacturing Part Number 82480-90016

Introducing the Math Pac

The Math Pac is a set of powerful tools for solving a wide range of mathematical, scientific, and engineering problems. These tools are provided in the convenient and flexible form of BASIC keywords. Once the math module is plugged into your HP-71 Computer, these keywords are instantly available: no program to load, no waiting. You can use these keywords in any program as often as needed; you avoid the restrictions that would apply to program calls and save the memory that subroutines would require.

The Math Pac adds the following capabilities to your HP-71.

- Complex variables and arrays.
- Advanced real- and complex-valued functions.
- Real and complex array operations.
- Solutions to systems of equations.
- Roots of polynomial equations and user-defined functions.
- Numerical integration.
- Finite Fourier transform.

Contents

How To Use This Manual

This manual assumes that you are generally familiar with the operation of your HP-71 Computer, especially how to create, edit, store, and run programs. You should also understand the mathematical basis for the operations you will be performing. Because the keywords in the Math Pac cover such a wide range of mathematical subjects, we cannot provide much tutorial information on the mathematical concepts involved.

The keywords in the Math Pac are independent of one another, so you may deal with only the keywords that specifically interest you. Each section in this manual contains information on keywords of a particular mathematical type—complex functions and operations, array arithmetic, and so on. All keywords described after section 5 (except FNROOT and INTEGRAL) use arrays in their operation. For an introduction to arrays, as used with the HP-71, read sections 3 and 14 of the *HP-71 Owner's Manual.*

Variable Declarations

The examples and programs in the Math Pac assume all variables are simple real unless otherwise declared. If an ERR: Data Type occurs as you execute an example or program, declare as REAL any variable not otherwise declared and continue operation.

Array Types

The Math Pac refers to two types of arrays, *vectors* and *matrices*. As used in this manual, the term vector identifies a singly-subscripted array, and matrix identifies a doubly-subscripted array. A subscript must be a real numeric expression. At run time, a subscript expression is rounded to an integer. The value of this integer must be in the range [0,65535] (OPTION BASE 0) or [1,65535] (OPTION BASE 1). Of course, in virtually all cases, available memory will determine the largest subscript you can use.

An array can be one of five data types: REAL, SHORT, INTEGER, COMPLEX, or COMPLEX SHORT (refer to section 3 for a description of COMPLEX and COMPLEX SHORT). Math Pac MAT statements will not change the declared type of an array; for example, when the values from a REAL array are assigned to a SHORT or INTEGER array, the values are rounded as they are stored into that array.

Array Redimensioning

Some Math Pac keywords allow you to *optionally* redimension an array. This is called *explicit* redimensioning. Other keywords *automatically* redimension result arrays, if possible, to accomodate the number of elements generated by the keyword's action. This is called *implicit* redimensioning. The kind of array redimensioning performed by a keyword, explicit or implicit, is stated in each keyword's description.

Explicit redimensioning occurs when an array's size and subscript count is changed according to the number and value of new subscripts supplied by you. For example, if **A** is a 3 × 4 REAL type matrix, then the HP-71 statement REAL **A**(3) *explicitly* redimensions **A** to be a 3 dimensional vector. Note that explicit redimensioning allows arrays to be changed from vectors to matrices and vice-versa. Explicit redimensioning also re-evaluates OPTION BASE; that is, resets the lower bound of an array's subscripts if the OPTION BASE setting has changed.

Implicit redimensioning occurs only in Math Pac operations of the form

MAT *result array* = *operation (operand array(s))*.

Implicit redimensioning only changes an array's size. It does not allow changes between vectors and matrices, nor does it re-evaluate OPTION BASE.

Keyword Description

Within each section you will find a description of each keyword name, function, syntax, and operation in the following format.

KEYWORD NAME **Function That the Keyword Performs**

Syntax Legal data types and numeric values for use with this keyword.
Description of the values returned by this keyword and the details of the keyword's operation.

Keyword Name. This is the way the keyword will be referenced elsewhere in the manual. It is usually a mnemonic of the function that the keyword performs. In most cases the name must be embedded in a longer statement that includes arguments, parentheses, and so on; the name by itself usually isn't an acceptable BASIC statement.

Several keywords have names that are identical to names of keywords already present in your HP-71—like DISP, +, and *. The syntax in which such a name is embedded indicates which operation to perform. All operations available to you in the HP-71 itself are still available, unaffected by the presence of the Math Pac.

Syntax. This is a description of the acceptable BASIC statements in which the keyword's name can be embedded. The following conventions are used throughout the manual in describing the syntax of a keyword.

Typographical Item	Interpretation
DOT MATRIX	Words in dot matrix (like COMPLEX) can be entered in lowercase or uppercase letters. The examples in this manual show statements, functions, and operators entered in UPPERCASE.
italic	Items in italics are the variables or parameters you supply, such as *X* in the SINH(*X*) statement.
bold	Variables in bold type represent arrays.
[]	Square brackets enclose optional items. For instance, MAT **A**=IDN[(*X*,*Y*)] indicates the redimensioning subscripts *X* and *Y* are optional.
stacked items	When items are placed one above the other, one and only one must be chosen.
...	An ellipsis indicates that the optional items within the brackets can be repeated. For instance, MAT INPUT **A**[,**B**]... indicates that MAT INPUT requires at least one array variable, and may accept several, with the array variables separated by commas.

Legal Data Types and Numeric Values. This information, in the same box as the syntax, describes the types and ranges of arguments for the keyword that are acceptable to the Math Pac. Use this information to avoid generating errors and to isolate the cause of those that do occur. *This is not a mathematical definition of the domain of the function that the keyword computes.*

Values Returned and Details of Operation. This information, in the box just below the syntax box, describes how the keyword works, tells what values the keyword returns, states whether array redimensioning (if any) is explicit or implicit, and states whether or not the keyword is usable in CALC mode.

Examples

Included in each section are a number of examples illustrating the use of the keywords in the section. To try an example yourself, type in the statements given in the **Input/Result** column using either upper- or lowercase, ending each line with with an [END LINE]. After you complete a line, the display of your HP-71 should look like the display shown in the **Input/Result** column following the line—provided that you have set your HP-71 operating conditions as indicated below.

- All operating conditions should be set as listed in the reference manual in the Systems Characteristics Section under the topic Reset Conditions, except for those whose settings follow.
- Set line width to 22 by entering `WIDTH 22` [END LINE].
- Set `DELAY` so that each display in a sequence of displays, often produced by a single statement, will remain visible long enough to be read and understood. The `DELAY` statement is described in The *HP-71 Reference Manual* and section 1 of the *HP-71 Owner's Manual.* In each you'll find descriptions of how you can control the length of time each display remains visible. For the display of array elements, you may find a `DELAY 0` setting useful. This causes each display to remain until any key, such as [END LINE], is pressed.

Additional Information

Some sections in the Math Pac include additional information to help you make effective use of the more sophisticated operations. If you would like still more information, you can refer to the *HP-15C Advanced Functions Handbook.* Although the Math Pac differs from the HP-15C Advanced Programmable Scientific Calculator in its operation and capabilities, much of the information in the *HP-15C Advanced Functions Handbook* applies to the Math Pac. Such information includes techniques to increase the effectiveness of equation-solving algorithms, integration algorithms, matrix operations, system solutions, and accuracy of numerical calculations.

Section 1

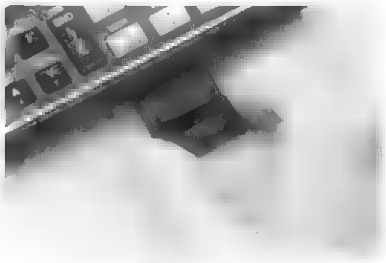
Installing and Removing the Module

The Math Pac module can be plugged into any of the four ports on the front edge of the computer.

CAUTIONS

- Be sure to turn off the HP-71 (press f OFF) before installing or removing the module.
- If you have removed a module to make a port available for the math module, before installing the math module, turn the computer on and then off to reset internal pointers.
- Do not place fingers, tools, or other objects into any of the ports. Such actions could result in minor electrical shock hazard and interference with pacemaker devices worn by some persons. Damage to port contacts and internal circuitry could also result.
- If a module jams when inserted into a port, it may be upside down. Attempting to force it further may result in damage to the computer or the module.
- Handle the plug-in modules very carefully while they are out of the computer. Do not insert any objects in the module connecter socket. Always keep a blank module in the computer's port when a module is not installed. Failure to observe these cautions may result in damage to the module or the computer.

To insert the Math Pac module, orient it so that the label is right-side up, hold the computer with the keyboard facing up, and push in the module until it snaps into place. During this operation be sure to observe the precautions described above.



To remove the module, use your fingernails to grasp the lip on the bottom of the front edge of the module and pull the module straight out of the port. Install a blank module in the port to protect the contacts inside.

Section 2

Base Conversions

Binary, Octal, and Hexadecimal Representations

The operations in this section allow your HP-71 to recognize and manipulate numbers expressed in number systems other than decimal (base 10).

Because the HP-71 assumes that any real number stored in a numeric variable or entered from the keyboard is a decimal number, you must enter and store every non-decimal number as a character string. In particular, if you store the number in a variable, the variable's name must end with "$"; if you enter the number from the keyboard, it must be enclosed in quotes.

In the tables below, *S$* will represent a binary, octal, or hexadecimal string or string expression.

- A *binary string* consists entirely of 0's and 1's, and represents a number in the base 2 number system. A *binary string expression* is a string expression whose value is a binary string.
- An *octal string* consists entirely of 0's, 1's, ..., 6's, and 7's, and represents a number in the base 8 number system. An *octal string expression* is a string expression whose value is an octal string.
- A *hexadecimal string* consists of 0's, ..., 9's, A's, ..., and F's (the letters may be either uppercase or lowercase), and this string represents a number in the base 16 number system. A *hexadecimal string expression* is a string expression whose value is a hexadecimal string.

Base Conversion Functions

BVAL — Binary, Octal, or Hexadecimal to Decimal Conversion

BVAL(*S$*,*N*)

where *S$* is a binary string expression whose value is not greater than 11101000110101001010010100001111111111111 (binary), and *N* is a numeric expression whose rounded integer value is 2;

or *S$* is an octal string expression whose value is not greater than 16432451207777 (octal), and *N* is a numeric expression whose rounded integer value is 8;

or *S$* is a hexadecimal string expression whose value is not greater than E8D4A50FFF (hexadecimal), and *N* is a numeric expression whose rounded integer value is 16.

BVAL (continued)

Converts a string expression *S$* representing ■ number expressed in base *N* into the equivalent decimal number. The value of the decimal equivalent can't exceed 999,999,999,999 (decimal).

Not usable in CALC mode.

BSTR$ — Decimal to Binary, Octal, or Hexadecimal Conversion

```
BSTR$(X,N)
```

where *X* is a numeric expression, $0 \leq X < 999{,}999{,}999{,}999.5$, and *N* is a numeric expression whose rounded integer value is 2, 8, or 16.

Converts the rounded integer value of *X* (decimal) into the equivalent base *N* string.

When *N* = 16, returns *uppercase* A, ..., F.

Not usable in CALC mode.

Examples

Input/Result

```
BVAL("1010",2) END LINE
```

```
 10
```

The decimal value of 1010 (binary).

```
B$="1111" END LINE
BVAL(B$,2) END LINE
```

```
 15
```

The decimal value of the binary string "1111."

```
BVAL(B$&B$,2) END LINE
```

```
 255
```

The decimal value of the binary string "11111111."

```
BSTR$(3,2) END LINE
```

```
 11
```

The binary representation of 3 (decimal).

```
BSTR$(72,8) END LINE
```

```
 110
```

The octal representation of 72 (decimal).

```
BSTR$(BVAL("AF1C8",16),2)
END LINE
```

```
 10101111000111001000
```

The binary representation of AF1C8 (hexadecimal).

```
BSTR$(BVAL("14772",8)
+BVAL("570",8),8) END LINE
```

```
 15562
```

The octal sum of 14772 (octal) and 570 (octal).

Additional Information

Three considerations determined the range of acceptable parameters for the base conversion keywords.

- The keywords give the exact answer for any integer in the range of acceptable parameters.
- The keywords are inverses of one another, so that composition in either direction is the identity transformation for integers.
- The integers from 0 through 999,999,999,999 form the largest block of consecutive non-negative integers that the HP-71 can display in integer format.

Section 3

Complex Variables

Complex Data Types

The operations in this section allow your HP-71 to declare, recognize and manipulate complex numbers. These operations include:

- Declaration of complex variables and arrays using COMPLEX and COMPLEX SHORT statements.
- Extension of HP-71 variable assignment and the RES function to the complex case.
- Extension of HP-71 IMAGE format strings to include complex fields
- Conversion of real numbers to complex.

Declaring Complex Variables

COMPLEX

Complex Variable Creation with 12-Digit Precision

COMPLEX *variable list*

where the syntax is the same as that used for REAL, SHORT, and INTEGER keywords. That is, each variable in the *variable list* has the form *numeric variable* [(*dim 1* [, *dim 2*])], and *dim 1* and *dim 2* are real numeric expressions.

Not usable in CALC mode.

COMPLEX SHORT

Complex Variable Creation with 5-Digit Precision

COMPLEX SHORT *variable list*

where the syntax is the same as that used for REAL, SHORT, and INTEGER keywords. That is, each variable in the *variable list* has the form *numeric variable* [(*dim 1* [, *dim 2*])], and *dim 1* and *dim 2* are real numeric expressions.

Not usable in CALC mode.

COMPLEX and COMPLEX SHORT both allocate memory for variables and arrays. If the array or variable does not already exist, creation occurs upon execution of the COMPLEX or COMPLEX SHORT statement, and all variables and array elements are initialized to (0,0). The dimension limits of arrays are evaluated at creation time. The lowest numbered subscript in any dimension is 0 or 1, depending upon the OPTION BASE setting when the array is created.

A COMPLEX statement redimensions existing arrays if they are type COMPLEX, but does not reinitialize them to (0,0). Similarly, a COMPLEX SHORT statement redimensions existing arrays if they are type COMPLEX SHORT, but does not reinitialize them to (0,0). If an array is being expanded, then all newly-created elements will be initialized. Redimensioning does preserve the sequence of elements within an array, but not necessarily the elements' positions within an array. Refer to the *HP-71 Owner's Manual*, section 3, under the topic *Declaring Arrays* (DIM, REAL, SHORT, INTEGER), for more information.

The following table indicates the conditions that apply to COMPLEX and COMPLEX SHORT variables and arrays.

COMPLEX and COMPLEX SHORT Numeric Variables

Initial value	(0, 0)
Numeric precision	
COMPLEX	12 decimal digits
COMPLEX SHORT	5 decimal digits
Exponent range	±499
Maximum number of array dimensions	2
Maximum dimension limit	65535
Simple variable memory usage (bytes)	
COMPLEX	25.5
COMPLEX SHORT	18.5
Array memory usage (bytes)	
COMPLEX	16 • (dim 1 − option base + 1) • (dim 2 − option base + 1) + 9.5
COMPLEX SHORT	9 • (dim 1 − option base + 1) • (dim 2 − option base + 1) + 9.5

Complex Number Operations

(,)

Real to Complex Conversion

(*X*,*Y*)

where *X* and *Y* are real- or complex-valued numeric expressions.

This is the way the HP-71 recognizes a complex number: as an ordered pair of real numbers. Since (*X*,*Y*) is defined as (real part of *X*, real part of *Y*), if either *X* or *Y* is complex, (*X*,*Y*) is not necessarily equivalent to $X + iY$.

Can be used in CALC mode.

REPT

Real Part of Complex Number

REPT(*Z*)

where *Z* is a real- or complex-valued numeric expression.

Returns the real part (first component) of *Z*. If *Z* is real, REPT(*Z*) = *Z*.

Can be used in CALC mode.

IMPT

Imaginary Part of Complex Number

IMPT(*Z*)

where *Z* is a real- or complex-valued numeric expression.

Returns the imaginary part (second component) of *Z*. If *Z* is real, IMPT(*Z*) = 0.

Can be used in CALC mode.

Other Complex Operations

The Math Pac allows extension of many operations of the HP-71 to the complex case. These include numeric functions such as SIN, ↑, etc., as described in section 5. Other extensions are the ability to assign values to complex variables created by a COMPLEX or COMPLEX SHORT statement, execution of the RES function when the last result is complex, and so on. In other words, when the Math Pac module is plugged in, the HP-71 can operate with complex numbers in much the same way that it operates with real numbers.

An important feature provided by the Math Pac is the extension of IMAGE format strings to include complex field specifiers. This extension is described below. Refer to the IMAGE keyword entry in the *HP-71 Reference Manual* for additional information on format strings.

C(,) — Complex Field in an IMAGE String

[*n*]C(*format string*)

where *n* is an optional multiplier.

Causes a complex expression in a DISP or PRINT output list to be formatted according to the *format string*. The real part is formatted first and the imaginary part second. On output, the number is enclosed in parentheses, with the real and imaginary parts separated by a comma. The comma is sent out when the second numeric field is encountered.

The *format string* may not include:

- A carriage control symbol (#).
- String fields.
- Imbedded complex format strings.

The *format string* must include *two and only two* numeric specifiers, but no special restrictions (other than those stated above) are placed on non-numeric specifiers.

Not usable in CALC mode.

Complex expressions in a DISP USING or PRINT USING output list may only be formatted by a complex field in the IMAGE list. Likewise, real expressions in a DISP USING or PRINT USING output list may not be formatted by a complex field in the IMAGE list.

Examples

COMPLEX, COMPLEX SHORT, (,), REPT, IMPT

Input/Result

`DESTROY ALL` END LINE

Insures that none of the variables and arrays in the following statements exist. If one did exist, it would not be initialized to (0,0) when the variable or array declaration statement is executed.

`COMPLEX Z, W1(3), V(7,7)` END LINE

Creates a complex variable, a complex vector, and a complex matrix. The variable `Z` and all elements of the arrays `W1` and `V` are initialized to (0,0).

`COMPLEX SHORT C(4,7), Y` END LINE

Creates a complex short array and a complex short variable. `Y` and all elements of `C` are initialized to (0,0).

`Z = (1,SQR(25))` END LINE

Assigns the complex number $1 + 5i$ to `Z`.

`Z` END LINE

```
(1,5)
```

The HP-71 representation of the complex number $1 + 5i$.

`V(6,5)=3` END LINE

Assigns the real number `3` to the complex array element `V(6,5)`.

`V(1,1);V(6,5)` END LINE

Displays two array element values.

```
(0,0)  (3,0)
```

Complex element `V(1,1)` was assigned `(0,0)` at its creation. Since the real number `3` was assigned to a complex element, it becomes the complex number `(3,0)`.

`Y=(REPT(1,2),REPT(3,4))` END LINE

Assigns (1,3) to `Y`, since (1,3) is `(REPT(1,2),REPT(3,4))`.

`Y` END LINE

Displays the complex number `Y`.

```
(1,3)
```

RES END LINE

Displays the value of the most recently executed or displayed numeric expresion, which in this case is complex.

```
 (1,3)
```

REPT(Y);IMPT(Y) END LINE

```
 1  3
```

Complex IMAGE Fields

Input/Result

```
5 STD @ COMPLEX Y
10 Y=(69.14,-12.7)
20 DISP USING 100; Y
30 DISP USING 200; Y,Y
40 DISP USING 300; Y,Y
50 DISP USING 400; Y,Y,Y
60 DISP USING "C(DDD,DDD)";Y
100 IMAGE C(2D.2D,4D.2D"i")
200 IMAGE C(4Z,XXX,4*),/,C(4Z,XXX4*)
300 IMAGE C(B,K"i"),X,C(^,4*.2DE)
400 IMAGE 3C(2(DDD,XX))
```

RUN

```
 (69.14, -12.70i)
 (0069   ,-*13)
 (0069,   -*13)
```

Line 100 IMAGE display.
Line 200 IMAGE display.

```
(E,-12.7i) (,-127.00E-
001)
```

Line 300 IMAGE display.

```
( 69   ,-13   )( 69   ,-1
3   )( 69   ,-13   )
```

Line 400 IMAGE display.

```
( 69,-13)
```

Line 60 display.

Section 4

Real Scalar Functions

Hyperbolic Functions

The functions SINH, COSH, and TANH (described below) are also defined for complex arguments. See section 5.

SINH — Hyperbolic Sine

SINH(*X*)

where *X* is a real-valued numeric expression, $|X| < 1151.98569368$

Can be used in CALC mode.

COSH — Hyperbolic Cosine

COSH(*X*)

where *X* is a real-valued numeric expression, $|X| < 1151.98569368$

Can be used in CALC mode.

TANH — Hyperbolic Tangent

TANH(*X*)

where *X* is a real-valued numeric expression.

Can be used in CALC mode.

ASINH — Inverse Hyperbolic Sine

ASINH(*X*)

where *X* is a real-valued numeric expression.

Can be used in CALC mode.

ACOSH — Inverse Hyperbolic Cosine

ACOSH(*X*)

where *X* is a real-valued numeric expression, $X \geq 1$.

Can be used in CALC mode.

ATANH — Inverse Hyperbolic Tangent

ATANH(*X*)

where *X* is a real-valued numeric expression, $-1 < X < 1$.

Can be used in CALC mode.

Other Functions Performing Calculations

GAMMA — Gamma Function

GAMMA(*X*)

where *X* is a real-valued numeric expression whose range is defined as follows:

X not equal to zero or a negative integer.

$-253 < X < 254.1190554375$.

Within the range $-263 < X < -253$, certain values of *X* cause GAMMA(*X*) to underflow as indicated by the graph of GAMMA(*X*).

For $X < -263$, $|\text{GAMMA}(X)| < \text{MINREAL}$, so GAMMA(*X*) will always underflow here.

GAMMA (continued)

If X equals a positive integer, GAMMA(X) = FACT($X-1$).
In general, GAMMA(X) = $\Gamma(X)$, defined for $X > 0$ as

$$\Gamma(X) = \int_0^\infty t^{X-1} e^{-t} dt$$

and defined for other values of X by analytic continuation.

Can be used in CALC mode.

LOG2

Base 2 Logarithm

LOG2(X)

where X is a real-valued numeric expression, $X > 0$.

$$\text{LOG2}(X) = \log_2(X) = \frac{\ln(X)}{\ln(2)}$$

Can be used in CALC mode.

SCALE10

Power of Ten Scaling

SCALE10(X,P)

where X is ■ real-valued numeric expression and P is ■ real numeric expression that must evaluate to an integer value.

Multiplies X by 10 raised to the power P by adding P to the exponent of X. You will find SCALE10 useful in preventing intermediate underflows and overflows in long chain calculations.

Can be used in CALC mode.

Integer Round

IROUND — Round to Integer

```
IROUND(X)
```

where *X* is a real-valued numeric expression.

Rounds *X* to an integer using the current OPTION ROUND setting.

Can be used in CALC mode.

Functions Providing Information

NAN$ — Not-a-Number Diagnostic Information

```
NAN$(X)
```

where *X* is a real-valued numeric expression.

Returns a string representing the error number contained in its NaN argument; that is, the number of the error that caused the NaN to be created. The string returned is of the same form as the number returned by the ERRN function (refer to the *HP-71 Reference Manual)*. However, the LEX identification number is 0 for all NaNs created by Math Pac functions since the Math Pac uses only HP-71 error messages when creating NaNs.

If *X* is not a NaN, then NAN$(X) returns a null string.

Not usable in CALC mode.

NEIGHBOR — Nearest Machine Number

```
NEIGHBOR(X,Y)
```

where *X* and *Y* are real-valued numeric expressions.

Returns the nearest machine-representable number to *X* in the direction toward *Y*. This is the machine successor (or predecessor) of *X* depending on the relative location of *Y*. You will find NEIGHBOR useful when you wish to evaluate a function in a local neighborhood of a given value.

Can be used in CALC mode.

TYPE

Expression Type and Dimension

TYPE(*X*)

where *X* is ■ real-, complex-, string-, or array-valued expression.

Returns an integer from 0 through 8 depending on the type and dimension of *X* as shown in the following table.

Except for string and array arguments, can be used in CALC mode.

X	TYPE(*X*)
Simple real (includes INTEGER, SHORT, and REAL simple variables.)	0
Simple complex (includes COMPLEX and COMPLEX SHORT simple variables.)	1
Simple string	2
INTEGER array	3
SHORT array	4
REAL array	5
COMPLEX SHORT array	6
COMPLEX array	7
String array	8

Examples

COSH, SINH, ATANH, ACOSH

Input/Result

COSH(0) END LINE

Hyperbolic cosine of a numeric constant.

```
 1
```

```
SINH(1/3+2^3) END LINE
```

Hyperbolic sine of a numeric expression.

```
 2080.1308825
```

```
X=9 END LINE
ATANH(1/SQR(X)) END LINE
```

Inverse hyperbolic tangent of a numeric expression with a numeric variable.

```
 .34657359028
```

```
ACOSH(COSH(200)) END LINE
```

Inverse hyperbolic cosine of a numeric expression.

```
 200
```

LOG2, IROUND

Input/Result

```
LOG2(2^17) END LINE
```

Logarithm (base 2) of a numeric expression.

```
 17
```

```
OPTION ROUND NEAR END LINE
```

Rounds to the nearest integer (the nearest even integer in case of a tie).

```
IROUND(234.5) END LINE
```

```
 234
```

```
OPTION ROUND POS END LINE
IROUND(234.5) END LINE
```

Rounds to the nearest larger integer.

```
 235
```

NAN$, NEIGHBOR, TYPE

Input/Result

```
X=TRAP(IVL,2) END LINE
```

Sets trap value 2 for IVL. Refer to the *HP-71 Reference Manual* for information on the TRAP function.

```
X=SIN(INF) END LINE
```

```
WRN: Invalid Arg
```

Trap value 2 for IVL causes a warning, not an error, to be given when the invalid operation SIN(INF) is executed.

```
X END LINE
```

```
NaN
```

The invalid operation assigns NaN (Not-a-Number) to X, since IVL has a trap value of 2.

```
NAN$(X) END LINE
```

```
11
```

The message number associated with the value NaN identifies the Invalid Arg message.

```
NEIGHBOR(1,5) END LINE
```

```
1.00000000001
```

The nearest machine number to 1 in the direction toward 5.

```
NEIGHBOR(1,-10) END LINE
```

```
.999999999999
```

The nearest machine number to 1 in the direction toward -10.

```
NEIGHBOR(1E400,1E401) END LINE
```

```
1.00000000001E400
```

The nearest machine number to 1E400 in the direction toward 1E401.

```
NEIGHBOR(1.234E-63,0) END LINE
```

```
 1.23399999999E-63
```

The nearest machine number to 1.234E-63 in the direction toward 0.

```
INTEGER I,J(3,9) END LINE
COMPLEX SHORT Z(2),W END LINE
TYPE(2);TYPE(I);TYPE(J);TYPE(Z)
;TYPE(W) END LINE
```

```
 0  0  3  6  1
```

The numbers returned by TYPE identify the type and dimension of each of the expressions.

Section 5

Complex Functions and Operations

Many useful functions are defined for complex as well as real arguments. The Math Pac allows you to use many HP-71 keywords for both complex and real arguments. In addition, this section describes other keywords defined specifically for complex operations.

All the functions and operations described in this section (except ABS, ARG, CONJ, and the relational operators) return a complex-type result.

With the exception of the RECT function, all complex numbers Z and W are assumed to be in rectangular, not polar, form.

The two-dimensional nature of these functions precludes giving simple bounds for the arguments that will avoid underflow and overflow messages.

Operators

+

Addition

$Z+W$

where Z and/or W are complex-valued numeric expressions.

Can be used in CALC mode.

−

Unary Minus

$-Z$

where Z is a complex-valued numeric expression.

Can be used in CALC mode.

− Subtraction

Z−W

where *Z* and/or *W* are complex-valued numeric expressions.

Can be used in CALC mode.

* Multiplication

Z*W

where *Z* and/or *W* are complex-valued numeric expressions.

Can be used in CALC mode.

/ Division

Z/W

where *Z* and/or *W* are complex-valued numeric expressions, $W \neq (0,0)$.

Can be used in CALC mode.

^ Exponentiation

Z^W

where *Z* and/or *W* are complex-valued numeric expressions.

Returns the principal value of $Z^W = e^{W\ln(Z)}$.
Can be used in CALC mode.

Logarithmic Functions

LOG — Natural Logarithm

LOG(Z) or LN(Z)

where Z is a complex-valued numeric expression, $Z \neq (0,0)$.

If $Z = x + iy$, and R (cos θ + i sin θ) is the polar representation of Z, then

$$\text{LOG}(Z) = \ln R + i\theta.$$

where $-\pi \leq \theta \leq \pi$ (radian measure).

Can be used in CALC mode.

EXP — Exponential

EXP(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{EXP}(Z) = e^{x + iy} = e^x (\cos y + i \sin y).$$

where y is taken to be radian measure.

Can be used in CALC mode.

Trigonometric and Hyperbolic Functions

All trigonometric calculations take their arguments to be in radian measure regardless of the angular setting.

SIN — Sine

SIN(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{SIN}(Z) = \sin(x + iy) = \sin x \cosh y + i \cos x \sinh y.$$

Can be used in CALC mode.

COS — Cosine

COS(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{COS}(Z) = \cos(x + iy) = \cos x \cosh y - i \sin x \sinh y.$$

Can be used in CALC mode.

TAN — Tangent

TAN(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{TAN}(Z) = \tan(x + iy) = \frac{\sin(x + iy)}{\cos(x + iy)} = \frac{\sin x \cos x + i \sinh y \cosh y}{\sinh^2 y + \cos^2 x}$$

Can be used in CALC mode.

SINH — Hyperbolic Sine

SINH(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{SINH}(Z) = \sinh(x + iy) = (-i)\sin(-y + ix).$$

Can be used in CALC mode.

COSH — Hyperbolic Cosine

COSH(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{COSH}(Z) = \cosh(x + iy) = \cos(-y + ix).$$

Can be used in CALC mode.

TANH — Hyperbolic Tangent

TANH(Z)

where Z is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{TANH}(Z) = \tanh(x + iy) = (-i)\tan(-y + ix).$$

Can be used in CALC mode.

Polar/Rectangular Conversions

POLAR

Rectangular to Polar Conversion

`POLAR(Z)`

where Z is a real- or complex-valued numeric expression.

If $Z = x + iy$, and $R(\cos\theta + i \sin\theta)$ is the polar representation of Z, then

$$\text{POLAR}(Z) = (R, \theta)$$

The angle θ is expressed in degrees ($-180 \leq \theta \leq 180$) or radians ($-\pi \leq \theta \leq \pi$) according to the current angular setting.

Can be used in CALC mode.

RECT

Polar to Rectangular Conversion

`RECT(Z)`

where Z is a real- or complex-valued numeric expression.

`RECT` is the only keyword in this section that assumes its argument Z to be in polar form.

If $Z = (R,\theta)$, where $R(\cos\theta + i \sin\theta)$ is the polar representation of the complex number $x + iy$, then

$$\text{RECT}(Z) = x + iy$$

The angle θ is taken to be in degrees or radians according to the current angular setting.

Can be used in CALC mode.

Squared Functions

SQRT

Square Root

`SQRT(Z)` or `SQR(Z)`

where Z is a complex-valued numeric expression.

Returns the complex principal value of the square root of Z.

Can be used in CALC mode.

SGN

Unit Vector

SGN(*Z*)

where *Z* is a complex-valued numeric expression.

Returns the unit vector in the direction of *Z*; that is,

$$\text{SGN}(Z) = \frac{Z}{|x + iy|} = \frac{x + iy}{\sqrt{x^2 + y^2}}$$

where $Z = x + iy$.

If $Z = (0,0)$, then SGN(*Z*) = *Z*.

Can be used in CALC mode.

ABS

Absolute Value

ABS(*Z*)

where *Z* is a complex-valued numeric expression.

If $Z = x + iy$, then

$$\text{ABS}(Z) = |x + iy| = \sqrt{x^2 + y^2}$$

ABS(*Z*) always returns real type.

Can be used in CALC mode.

ARG

Argument

`ARG(Z)`

where *Z* is a real- or complex-valued numeric expression.

If $Z = x + iy$ and $R(\cos\theta + i\sin\theta)$ is the polar representation of *Z*, then

`ARG(Z)` $= \theta$.

The angle θ is expressed in degrees ($-180 \leq \theta \leq 180$) or radians ($-\pi \leq \theta \leq \pi$) according to the current angular setting.

`ARG(Z)` always returns real type.

Can be used in CALC mode.

CONJ

Complex Conjugate

`CONJ(Z)`

where *Z* is a real- or complex-valued numeric expression.

If $Z = x + iy$, then

`CONJ(Z)` $= x - iy$

`CONJ(Z)` always returns the same type (real or complex) as *Z*.

Can be used in CALC mode.

PROJ

Projective Infinity

`PROJ(Z)`

where *Z* is a real- or complex-valued numeric expression.

If $Z = x + iy$, then

`PROJ(Z)` $= Z$ if `ABS(Z)` $\neq$ `Inf`

or

`PROJ(Z)` $=$ `Inf` $+ i0$ if `ABS(Z)` $=$ `Inf`.

Can be used in CALC mode.

Relational Operators

=, <, >, #, ?

Equal or Unordered

Z comparison operator W

where *Z* and/or *W* are complex-valued numeric expressions.

When at least one of two expressions is complex valued, only two comparison results are possible: either the expressions are equal or they are unordered (or unequal, which is equivalent to unordered in this case).

Suppose $Z = x + iy$ and $W = u + iv$.

If $x = u$ and $y = v$, then any comparison that contains = is true (that is, evaluates to 1).

If $x \neq u$ or $y \neq v$, then any comparison that contains # or ? is true.

Any comparison that contains < or > without ? or # produces an exception.

Can be used in CALC mode.

Examples

+, −, *, /

Input/Result

```
STD @ COMPLEX Z,W END LINE
Z=(4,5) @ W=(-3,2) END LINE
Z+W END LINE
```

```
(1,7)
```

```
3+Z+W+1 END LINE
```

```
(5,7)
```

```
Z-W END LINE
```

```
(7,3)
```

(2,3)*(4,5) END LINE

```
(-7,22)
```

(1,2)/(3,4) END LINE

```
(.44,.08)
```

2/(3,4) END LINE

```
(.24,-.32)
```

^, LOG, EXP

Input/Result

FIX4 END LINE
(3,4)^(6,9) END LINE

```
(1.3472,3.4565)
```

LOG((1,2)) END LINE

```
(0.8047,1.1071)
```

EXP((1,2)) END LINE

```
(-1.1312,2.4717)
```

SIN, TAN, COSH

Input/Result

```
FIX4 [END LINE]
SIN((21,2)) [END LINE]
```

```
     (3.1477,-1.9865)
```

```
TAN((5,5)) [END LINE]
```

```
     (-4.9401E-5,1.0001)
```

```
COSH((2,3)) [END LINE]
```

```
     (-3.7245,0.5118)
```

ABS, ARG, CONJ, PROJ

Input/Result

```
FIX4 [END LINE]
ABS((3,4)) [END LINE]
```

```
     5.0000
```

```
DEGREES [END LINE]
ARG((3,4)) [END LINE]
```

```
     53.1301
```

```
RADIANS [END LINE]
ARG((3,-7)) [END LINE]
```

```
    -1.1659
```

The fourth quadrant angle θ measured in radians, which is the argument of the complex number $3 - 7i$.

```
STD @ CONJ((1,2)) END LINE
```

```
(1,-2)
```

```
PROJ((-Inf,-Inf)) END LINE
```

```
(Inf,0)
```

```
PROJ((1,2)) END LINE
```

```
(1,2)
```

POLAR, RECT, SGN

Input/Result

```
STD END LINE
DEGREES END LINE
POLAR(-1) END LINE
```

Rectangular to polar conversion for a real argument.

```
(1,180)
```

The absolute value (r) is 1 and the argument (θ) is 180 degrees.

```
FIX4 END LINE
POLAR((3,4)) END LINE
```

Rectangular to polar conversion for a complex argument.

```
(5.0000,53.1301)
```

The absolute value (r) is 5.0000 and the argument (θ) is 53.1301 degrees.

```
RADIANS END LINE
RECT((-5,PI/4)) END LINE
```

Polar to rectangular conversion for a complex argument. The absolute value (r) is 5 and the argument (θ) is $-3\pi/4$ radians. Since the R given is negative, this is the reflection of the polar point (5,PI/4) through the origin.

```
(-3.5355,-3.5355)
```

The real part (x) and the imaginary part (y) are both -3.5355.

```
SGN((1,1)) END LINE
```

```
(0.7071,0.7071)
```

SQRT, LOG

Note the behavior of SQRT and LOG at the branch cut. Refer to the discussion of branches under the "Additional Information" topic below.

Input/Result

```
FIX4 @ SQRT((1,2)) END LINE
```

```
(1.2720,0.7862)
```

```
SQRT((-16,0)) END LINE
```

```
(0.0000,4.0000)
```

```
SQRT((-16,-0)) END LINE
```

```
(0.0000,-4.0000)
```

```
LOG((-EXP(5),0)) END LINE
```

```
(5.0000,3.1416)
```

```
LOG((-EXP(5),-0)) END LINE
```

```
(5.0000,-3.1416)
```

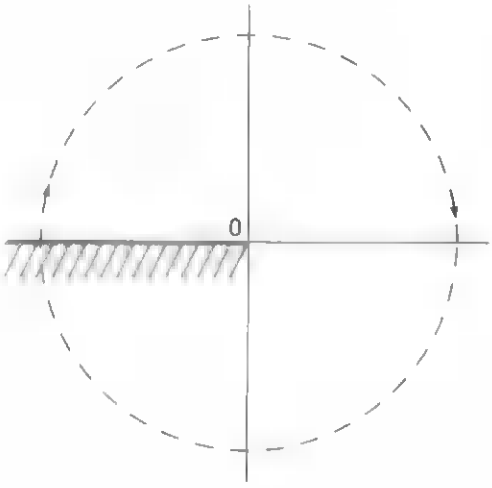
Additional Information

In general, the inverse of a function $f(z)$—denoted $f^{-1}(z)$—has more than one value for any argument z. However, the Math Pac calculates the single *principal value*, which lies in the part of the range defined as the *principal branch* of the inverse function $f^{-1}(z)$.

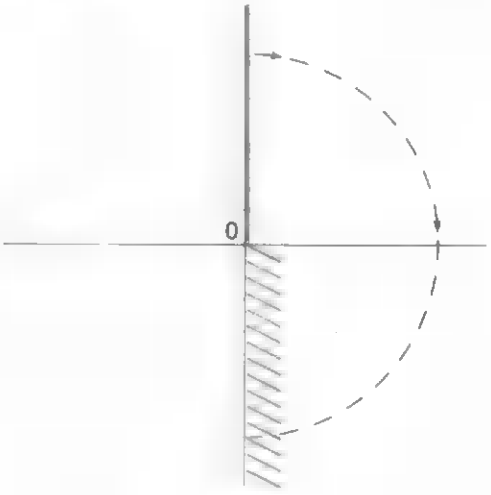
The illustrations that follow show the principal branches that the Math Pac uses for SQRT and LOG. The left-hand graph in each figure represents the cut domain of the inverse function; the right-hand graph shows the range of the principal branch. The blue and the black lines in the left-hand graph are mapped, under the inverse function, to the corresponding blue and black lines in the right-hand graph.

SQRT

$$\sqrt{z} = \sqrt{r}\, e^{i\theta/2} \text{ for } -\pi \leq \theta \leq \pi$$



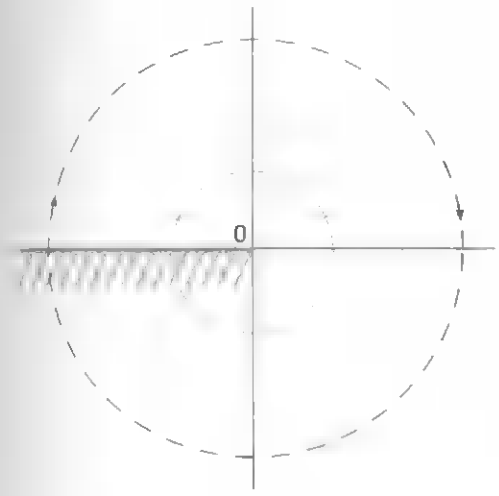
z



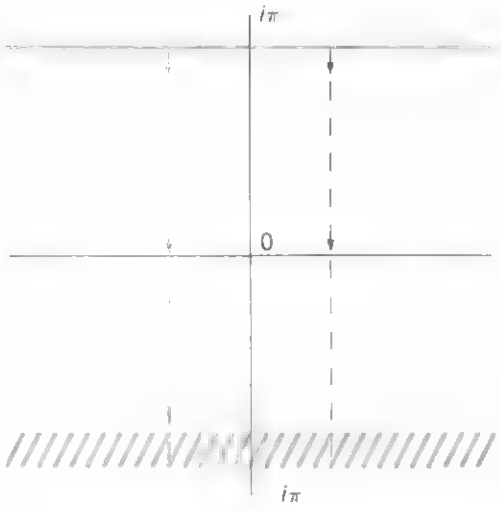
$w = \sqrt{z}$

LOG

LN(z) = ln r + iθ for $-\pi \leqslant \theta \leqslant \pi$



z



w = LN(z)

The principal branch of w^z is derived from that of the log function and the equation:

$$w^z = \exp(z \text{ LN } w),$$

where LN denotes the single-valued function.

To determine *all* values of the inverse function, use the expressions below to derive these values from the principal values calculated by the Math Pac. In these expressions, $k = 0, \pm 1, \pm 2$, and so on, and uppercase letters denote single-valued functions.

$\sqrt{z} = \pm \text{SQR}(z)$ $\qquad \ln(z) = \text{LN}(z) + 2\pi ik$ $\qquad w^z = w^z e^{2\pi ikz}$

Section 6

Array Input and Output

The keywords in this section enable you to:

- Fill an array with values.
- Display or print values already in an array.

Assignments

=

Simple Assignment

MAT **A**=**B**

where **A** and **B** are both vectors or both matrices.
Array **B** may be either real or complex type.
If **B** is complex, then **A** must be complex.
If **B** is real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B**, and assigns the value of every element in **B** to the corresponding element in **A**.

To halt operation, press ATTN twice.

Not usable in CALC mode.

= ()

Numeric Expression Assignment

```
MAT A=(X)
```

where *X* is either a real- or complex-valued numeric expression.
If *X* is complex, then array **A** must be complex type.
If *X* is real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Assigns *X* to all elements of **A**. Array **A** is not redimensioned.

To halt operation, press ATTN twice.

Not usable in CALC mode.

CON

Constant Array

```
MAT A=CON [(X [,Y] )]
```

where **A** is either a real- or complex-type array, and the optional redimensioning subscripts *X* and *Y* are real-valued numeric expressions. *X* and *Y* are rounded to the nearest integer just as are subscripts in DIM statements.

Assigns the real value one to all elements of **A**. If redimensioning subscript(s) are provided, **A** is explicitly redimensioned according to the number and value of those subcripts.

Not usable in CALC mode.

IDN

Identity Matrix

```
MAT A=IDN [(X,Y)]
```

where **A** is a real- or complex-type array and where the optional redimensioning subscripts *X* and *Y* are real-valued numeric expressions with the same rounded integer value. *X* and *Y* are rounded to the nearest integer just as are subscripts in DIM statements. If *X* and *Y* are not provided, **A** must be a square matrix (it must have two equal subscripts).

If no redimensioning subscripts *X* and *Y* are provided, then **A** will become an identity matrix. If redimensioning subscripts *X* and *Y* are provided, then **A** is explicitly redimensioned to a square matrix with the upper bound of each subscript equal to the rounded integer value of *X* and *Y* and then assigned the values of an identity matrix.

Not usable in CALC mode.

ZER

Zero Array

MAT **A**=ZER [(*X* [,*Y*])] or MAT **A**=ZERO [(*X* [,*Y*])]

where **A** is either a real- or complex-type array, and the optional redimensioning subscripts *X* and *Y* are real-valued numeric expressions. *X* and *Y* are rounded to the nearest integer just as are subscripts in DIM statements.

Assigns zero to all elements of **A**. If redimensioning subscript(s) are provided, **A** is explicitly redimensioned according to the number and value of those subcripts.

Not usable in CALC mode.

Array Input

INPUT

Assign Values from Keyboard Input

MAT INPUT **A** [,**B**]...

where **A** (and **B**) are real- or complex-type array(s).

Assigns real or complex numbers to the specified array(s). Complex values cannot be assigned to real array elements. MAT INPUT prompts with the name of an array element and then accepts a numeric expression from the keyboard, evaluates that expression, and assigns the result as the value of that element. For each array, MAT INPUT gives prompts for the elements in row order (from left to right in each row, from the first row to the last). If there is more than one array, they are handled in the order specified.

When the name of an array element is displayed, enter its value by typing in the numeric expression and then pressing [END LINE]. You can enter values for several consecutive elements by separating the values with commas. When an array is filled, the remaining values are automatically entered into the next array. After you press [END LINE], the computer will display the name of the next element (if any) to be assigned a value.

INPUT (continued)

In other respects, MAT INPUT acts as does INPUT. For instance:

- The Command Stack is always active during MAT INPUT execution. You can move up and down in the Command Stack with [▲], [▼], [g] [▲], and [g] [▼] without first pressing [g] [CMDS].
- You can use ■ direct execute user-defined key to provide the response to the MAT INPUT prompt.
- The [f] [VIEW] key sequence and the [g] [ERRM] key sequence are active during MAT INPUT execution.
- If you are making ■ response to a MAT INPUT statement, but have not pressed [END LINE], pressing [ATTN] once clears the typed entry, allowing another entry to be typed. If you press [ATTN] twice, the HP-71 clears the entry, pauses the program, and clears the display.

Not usable in CALC mode.

Array Output

To halt the operation of any of the keywords described below you need press [ATTN] only once.

DISP

Display in Standard Format

MAT DISP **A** [, **B**]...[,]
; ; ;

where **A** (and **B**) are real- or complex-type array(s).

Displays the values of the elements of the specified arrays. The values are displayed in row order. Each row begins on a new line; a blank line is displayed between the last row of an array and the first row of the next array.

The choice of terminator—comma or semicolon—determines the spacing between the elements of an array.

Terminator	Spacing Between Elements
,	Close: Elements are separated by two spaces. A minus sign, if present, occupies one of the two spaces.
;	Wide: Elements are placed in 21-column fields.

If the last array specified doesn't have ■ terminator, the array will be displayed with wide spacing between elements.

Not usable in CALC mode.

PRINT

Print in Standard Format

MAT PRINT **A** [, / ; **B**] ... [, / ;]

where **A** (and **B**) are real- or complex-type array(s).

Prints the values of the specified arrays. Operation is identical to MAT DISP, except that the output is sent to the PRINTER IS device, which requires HP-IL. If no PRINTER IS device is present, output is sent to the display, or to the HP-IL DISPLAY IS device. Also, you can override the CR/LF normally generated by MAT PRINT with the ENDLINE statement. ENDLINE is described in the *HP-71 Reference Manual* and in section 13 of the *HP-71 Owner's Manual*.

Not usable in CALC mode.

DISP USING

Display Using Custom Format

MAT DISP USING *format string* / *line number* ; **A** [, / ; **B**] ... [, / ;]

where **A** (and **B**) are real- or complex-type array(s).

Displays the values of the elements of the specified arrays in a format determined by the *format string* or by the specified IMAGE statement identified by the *line number*. (Refer to the *HP-71 Reference Manual* for information about DISP USING, format strings, IMAGE statements, and their results).

If any array is complex type, the corresponding field specifier in the *format string* or IMAGE statement must be a complex field specifier. Refer to the description of the complex field specifier (C(,)) in section 3, page 22.

The values are displayed in row order. Each row begins on a new line; a blank line is displayed between the last row of an array and the first row of the next array.

The terminators between the arrays—commas or semicolons—serve only to separate the arrays and have no effect on the display format.

The Math Pac must be plugged in to RENUMBER a program containing a MAT DISP USING [*line number*] statement; otherwise, the *line number* will not be correctly updated.

Not usable in CALC mode.

PRINT USING

Print Using Custom Format

MAT PRINT USING *format string* / *line number* ; **A** [, / ; **B**] ... [, / ;]

where **A** (and **B**) are real- or complex-type array(s).

Operation is identical to MAT DISP USING, except that the output is sent to the PRINTER IS device which requires HP-IL. If no PRINTER IS device is present, output is sent to the display, or to the HP-IL DISPLAY IS device. Also, you can override the CR/LF normally generated by MAT PRINT USING with the ENDLINE statement. ENDLINE is described in the *HP-71 Reference Manual* and in section 13 of the *HP-71 Owner's Manual*.

Not usable in CALC mode.

Examples

With the optional delay of 0 or larger (infinite line replacement delay), you press END LINE (or any other key) to display the next line. So you can control how long each array row is displayed.

CON, IDN, ZER, DISP

Input/Result

```
OPTION BASE 1 @ STD
DIM A(3,3),B(1)
COMPLEX C(10,20)
MAT A=IDN
MAT DISP A;
```

B is dimensioned to be a one element vector.

Displays the identity matrix **A** with close spacing between the elements.

```
 1  0  0
 0  1  0
 0  0  1
```

```
MAT B=ZER(2,2) END LINE
```

Redimensions **B** from a one-element vector to a 2 × 2 matrix and assigns to it a zero array.

```
MAT DISP B; END LINE
```

```
 0  0
 0  0
```

```
MAT C=CON(3,3) END LINE
```

Redimensions **C** and assigns to it a constant array.

```
MAT DISP C; END LINE
```

```
 (1,0)  (1,0)  (1,0)
 (1,0)  (1,0)  (1,0)
 (1,0)  (1,0)  (1,0)
```

INPUT

Input/Result

```
OPTION BASE 1 END LINE
DIM A(2,3),B(3) END LINE
OPTION BASE 0 END LINE
COMPLEX C(2,1) END LINE
```

Declares **C** to be a 3 × 2 complex array (remember we are in `OPTION BASE 0`).

```
MAT INPUT A,B,C END LINE
```

```
A(1,1)? ■
```

Prompts for the first element's value.

```
1,2,3,4 END LINE
```

More than one value can be entered.

```
A(2,2)? ■
```

Prompts for the fifth element's value.

```
5,6,7 END LINE
```

Enters values for the last two elements of **A** and the first element of **B**.

```
B(2)? ■
```

```
8,9,10 END LINE
```

Enters values for the last two elements of **B** and the first element of the complex array **C**.

```
C(0,1)? ■
```

```
1,2,(5,6),(7,8) END LINE
```

Enters values for the next four elements of **C**.

```
C(2,1)? ■
```

```
NAN END LINE
```

Enters "not a number" for the last element of **C**.

```
STD @ MAT DISP A;B;C; END LINE
```

Displays each array in sequence, with a blank line between each.

```
 1   2   3
 4   5   6

 7
 8
 9

(10,0)   (1,0)
(2,0)    (5,6)
(7,8)    (NaN,0)
```

DISP USING

Input/Result

```
10 OPTION BASE 1 @ INTEGER A(5,5)
15 WIDTH 22 @ DELAY 8
```

Causes the output to appear in the display as shown below. After each display, press [END LINE] to produce next display.

```
20 COMPLEX SHORT Z(3,4)
25 MAT A=IDN @ MAT Z=(4,5)
```

Assigns the identity matrix to **A** and the complex number (4,5) to every element of **Z**.

```
30 MAT DISP USING 'DDD,ZZZ';A,A
```

This format string consists of two field specifiers, DDD and ZZZ. Each element of **A** is displayed according to these field specifiers used repeatedly until all elements have been displayed. The final element of **A** is displayed according to DDD. Then a blank line is displayed, followed by another display of all elements of **A**. The field specifier ZZZ (the next specifier in the format string) is used to format the display of the first element during this second display of **A**.

```
35 MAT DISP USING '#,D';A @ DISP 4
```

The # symbol supresses the automatic end-of-line sequence (CR/LF) following the display of **A**. This causes 4 to be displayed on the same line as the last element of **A**.

```
40 MAT DISP USING 100;Z
45 DELAY 1
100 IMAGE C(K,2D,'i')
```

The IMAGE statement must use the C(,) form to format the display of a complex array. The parentheses must contain two numeric field specifiers.

RUN

```
    1000  0000  0
```

The □ format symbol replaces leading zeros with blanks. Since **A** is an identity matrix, element (1,1) is 1. Therefore the two leading zeros are replaced with blanks, and element (1,1) is displayed as 1. The Z format symbol fills each leading zero with 0, so element (1,2) is displayed as 000. The remaining elements, in row order, are displayed according to the format string □□□, ZZZ used repeatedly.

```
000  1000  0000
```

After the last (fifth) element of the first row is displayed, an end-of-line sequence (carriage return, line feed) is sent, causing the display of element (2,1) to start a new line.

```
  0000  1000  0
000  0000  1000
  0000  0000  1
```

The field specifier □□□ formats the display of the last element of **A**, causing the display of 1.

```

```

Following the display of the last element of the last row, a second end-of-line sequence is sent, causing the display of a blank line between the two displays of array **A**.

```
001  0000  0000
```

Since the variable list following the format string in line 30 is A,A, array **A** is displayed twice. This time, element (1,1) is displayed according to the field specifier ZZZ, since □□□ was used just above for the last element of **A** during the first display of this array.

```
  0001  0000  0
000  0001  0000
  0000  0001  0
000  0000  0001
```

Since this is the display of the *last* array in the variable list of line 30, no blank line is displayed, even though this display line ends with the last element of the last row of **A**.

```
10000
```

Since the portion of the format string of line 35 that controls character display consists only of D, the elements of each row of **A** are displayed with no extra characters or spaces.

```
01000
00100
00010
00001  4
```

The # symbol in the format string of line 35 supresses the end-of-line sequence normally sent after the display of the final row of the last array in the variable list.

```
(4, 5i)(4, 5i)(4, 5i)(
```

The symbol K in the format string of line 100 specifies a compact field, resulting in the display of no leading or trailing blanks. This symbol controls the display format of the real part of each (identical) element of **Z**. The display of the imaginary part of each element is controlled by 2D. Since the imaginary part, 5, consists of only one digit, a leading blank is displayed. The complex image specification D(,) causes the display of the parentheses and comma.

```
4, 5i)
```

The display of each row is ended with an end-of-line sequence, so each new row starts a new display line.

```
(4, 5i)(4, 5i)(4, 5i)(
4, 5i)
(4, 5i)(4, 5i)(4, 5i)(
4, 5i)
```

Section 7

Array Arithmetic

The keywords in this section perform arithmetic operations on arrays. The dimensions of the operand arrays must be compatible with the particular operation, as discussed below.

- For addition and subtraction, the operand arrays must both be vectors or both be matrices, and they must have the same number of rows and the same number of columns. In this case we will say that the arrays are *conformable for addition.*
- For multiplication of two arrays, the first array must be a matrix, while the second array can be a matrix or a vector. The number of *columns* of the first array must be equal to the number of *rows* of the second array. If these conditions are satisfied, we will say that the arrays are *conformable for multiplication.*
- For transpose multiplication of two arrays, the first array must be a matrix, while the second array can be a matrix or a vector. The number of *rows* of the first array must be equal to the number of *rows* of the second array. If these conditions are satisfied, we will say that the arrays are *conformable for transpose multiplication.*

Operators

=− **Negation**

MAT **A**=−**B**

where **A** and **B** are both vectors or both matrices.
Array **B** may be either real or complex type.
If **B** is complex, then **A** must be complex.
If **B** is real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B** and assigns to each element of **A** the negative of the corresponding element of **B**.

To halt operation, press [ATTN] twice.

Not usable in CALC mode.

+ Addition

MAT **A=B+C**

where **A**, **B**, and **C** are all vectors or all matrices, and **B** and **C** are conformable for addition.
Arrays **B** and **C** may be either real or complex type.
If either **B** or **C** is complex, then **A** must be complex.
If both **B** and **C** are real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B** and **C**, and assigns to each element of **A** the sum of the values of the corresponding elements of **B** and **C**.

To halt operation, press [ATTN] twice.

Not usable in CALC mode.

− Subtraction

MAT **A=B−C**

where **A**, **B**, and **C** are all vectors or all matrices, and **B** and **C** are conformable for addition.
Arrays **B** and **C** may be either real or complex type.
If either **B** or **C** is complex, then **A** must be complex.
If both **B** and **C** are real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B** and **C**, and assigns to each element of **A** the difference of the values of the corresponding elements of **B** and **C**.

To halt operation, press [ATTN] twice.

Not usable in CALC mode.

( )*

Multiplication by ■ Scalar

```
MAT A=(X)*B
```

where **A** and **B** are both vectors or both matrices and *X* is a numeric expression.
Array **B** may be either real or complex type and expression *X* may be either real or complex valued.
If either ■ or *X* is complex, then **A** must be complex.
If both **B** and *X* are real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B** and assigns to each element of **A** the product of the value of *X* and the value of the corresponding element of **B**.

To halt operation, press ATTN twice.

Not usable in CALC mode.

*

Matrix Multiplication

```
MAT A=B*C
```

where **B** is a matrix, **A** and **C** are both vectors or both matrices, and **B** and **C** are conformable for multiplication.
Arrays **B** and **C** may be either real or complex type.
If either **B** or **C** is complex, then **A** must be complex.
If both **B** and **C** are real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to have the same number of rows as **B** and the same number of columns as **C**. The values of the elements of **A** are determined by the usual rules of matrix multiplication.

To halt operation, press ATTN twice.

Not usable in CALC mode.

TRN *

Transpose Multiplication

```
MAT A= TRN(B)*C
```

where **B** is a matrix, **A** and **C** are both vectors or both matrices, and **B** and **C** are conformable for transpose multiplication.
Arrays **A** and **C** may be either real or complex type.
If either **B** or **C** is complex, then **A** must be complex.
If both **B** and **C** are real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to have the same number of rows as the number of columns in **B** and the same number of columns as **C**.

The result of this operation is the same as if the transpose of **B** (or the conjugate transpose of **B**, if **B** is complex type) was computed and then post-multiplied by **C**. However, the Math Pac uses special multiplication rules so that **B** does not have to be explicitly transposed prior to the multiplication.

To halt operation, press ATTN twice.

Not usable in CALC mode.

Examples

+, *, ( )*, TRN *

Input/Result

```
OPTION BASE 1 @ STD END LINE
REAL A(2,3),B(3,4) END LINE
COMPLEX SHORT C(3,1),D(2),E(9)
END LINE
MAT A=IDN(2,2) END LINE
MAT C=((3,4))*A END LINE
```

C is redimensioned to 2 × 2 and every element of **C** is assigned the product of the complex number (3,4) and the corresponding element of **A**.

```
MAT DISP C; END LINE
```

```
 (3,4)   (0,0)
 (0,0)   (3,4)
```

The array **C**.

```
MAT A=CON @ MAT C=C+A END LINE
```

C holds the array sum of **A** and **C**. No redimensioning is necessary since **C** is already the correct size.

```
MAT DISP C; END LINE
```

```
 (4,4)   (1,0)
 (1,0)   (4,4)
```

The array **C**.

```
MAT B=A*A END LINE
```

B is redimensioned to 2 × 2 to hold the matrix product **A*A**.

```
MAT DISP B; END LINE
```

```
 2   2
 2   2
```

The array **B**.

```
MAT INPUT D END LINE
```

```
D(1)? ■
```

```
(1,2),(3,4) END LINE
MAT E=TRN(C)*D END LINE
```

E is redimensioned to be a 2 element vector to hold the product of the conjugate transpose of **C** and the vector **D**.

```
MAT DISP E END LINE
```

```
 (15,8)
 (29,6)
```

The array **E**.

Section ■

Scalar-Valued Array Functions

The keywords in this section are functions that use real- or complex-type arrays as arguments (except DET uses only real arrays) and give a real number as ■ value (except DOT can give either a real or complex number). Like other HP-71 functions, they may be used alone or in combination with other functions to produce numeric expressions.

Determinant Functions

DET — Determinant

DET(**A**)

where **A** is ■ square real-type matrix.

Returns the determinant of the matrix **A**.

To halt operation, press [ATTN] twice.

Not usable in CALC mode.

DETL — Determinant of Last Matrix

DETL or DET

Returns the determinant of the last real-type matrix that was:

- Inverted in a MAT . . . INV statement (described in section 9).
- Used as the first argument of a MAT . . . SYS statement (described in section 9).

DETL retains its value (even if the HP-71 is turned off) until another MAT . . . INV (with a real type argument) or a MAT . . . SYS (with ■ real type first argument) is executed.

Not usable in CALC mode.

Array Norms

CNORM

One-Norm (Column Norm)

CNORM(**A**)

where **A** is a real- or complex-type array.

Returns the maximum value (over all columns of **A**) of the sums of the absolute values of all elements in a column. Refer to the keyword description for ABS, page 41 in section 5, for the definition of the absolute value of a complex number.

To halt operation, press ATTN twice.

Not usable in CALC mode.

RNORM

Infinity Norm (Row Norm)

RNORM(**A**)

where **A** is a real- or complex-type array.

Returns the maximum value (over all rows of **A**) of the sums of the absolute values of all elements in a row. Refer to the keyword description for ABS, page 41 in section 5, for the definition of the absolute value of a complex number.

To halt operation, press ATTN twice.

Not usable in CALC mode.

FNORM

Frobenius (Euclidean) Norm

FNORM(**A**)

where **A** is a real- or complex-type array.

Returns the square root of the sum of the squares of the absolute values of all elements of **A**. Refer to the keyword description for ABS, page 41 in section 5, for the definition of the absolute value of a complex number.

To halt operation, press ATTN twice.

Not usable in CALC mode.

Inner Product

DOT — Inner (Dot) Product

DOT(**X**, **Y**)

where **X** and **Y** are real- or complex-type vectors with the same number of elements.

Returns **X**•**Y**, the inner product of **X** and **Y**. If both **X** and **Y** are real, then the result is real. If either **X** or **Y** is complex, then the result is complex.

If **X** is a complex vector, then the complex conjugates of the elements of **X** are used to compute the inner product.

To halt operation, press [ATTN] twice.

Not usable in CALC mode.

Subscript Bounds

The following functions are useful in keeping track of array option base, number of dimensions, and size in each dimension, since these quantities may change when variables are dimensioned and redimensioned.

UBND — Subscript Upper Bound

UBND(**A**, *N*) or UBOUND(**A**, *N*)

where **A** is a real- or complex-type array and *N* is a numeric expression whose rounded integer value is 1 or 2.

Returns the upper bound of the *N*th (first or second) subscript of **A**. If **A** is a vector, UBND(A, 2) = −1.

Not usable in CALC mode.

LBND

Subscript Lower Bound

`LBND(`**A**`,`*N*`)` or `LBOUND(`**A**`,`*N*`)`

where **A** is a real- or complex-type array and *N* is a numeric expression whose rounded integer value is 1 or 2.

Returns the `OPTION BASE` setting in effect when **A** was last dimensioned. If **A** is a vector, `LBND(A,2)` = −1.

Not usable in CALC mode.

Examples

DET, DOT

Input/Result

`OPTION BASE 1` END LINE

`DIM A(10,10)` END LINE

`MAT A=IDN` END LINE

`MAT A=(-3)*A` END LINE

Assigns −3 to each diagonal element; all other elements remain zero.

`DET(A)` END LINE

Displays the determinant of **A**.

```
     59049
```

`MAT A=IDN(3,3)` END LINE

`MAT A=(2)*A` END LINE

Assigns 2 to each diagonal element; all other elements remain zero.

`MAT A=INV(A)` END LINE

Computes the inverse of **A**.

`DET` END LINE

Displays the determinant of the last real matrix inverted in a `MAT...INV` statement or used as the first argument of a `MAT...SYS` statement. Refer to pages 77-79 in section 9 for definitions of `INV` and `SYS`.

```
 8
```

```
DIM A(10),B(10) END LINE
MAT A=(2) END LINE
```
Assigns 2 to each element of **A**.

```
MAT B=CON END LINE
```
Assigns one to each element of **B**.

```
DOT(A,B) END LINE
```
Displays the inner product of **A** and **B**.

```
 20
```

```
COMPLEX C(10) END LINE
MAT C=(1,2) END LINE
```
Assigns the complex number (1,2) to each element of **C**.

```
DOT(C,A) END LINE
```
Displays the inner product (a complex number) of **C** and **A**.

```
 (20,-40)
```

RNORM, CNORM, FNORM, UBND, LBND

Input/Result

```
OPTION BASE 1 END LINE
DIM A(3,5) END LINE
MAT A=CON END LINE
```
Assigns 1 to each element of **A**.

```
RNORM(A) END LINE
```
Displays the row norm of **A**.

```
 5
```

```
COMPLEX SHORT  A(2,4) END LINE
MAT INPUT A END LINE
```

```
    A(1,1)? ■
```

```
(1,2),(3,4),(5,6),(7,8),(9,10)
(11,12),(13,14),(15,16)
END LINE
```

```
RNORM(A) END LINE
```

Displays the row norm of **A**.

```
    70.7691300172
```

```
CNORM(A) END LINE
```

Displays the column norm of **A**.

```
    32.5618580122
```

```
FNORM(A) END LINE
```

Displays the Frobenius norm of **A**.

```
    38.6781592117
```

```
COMPLEX B(3) END LINE
UBND(A,1);UBND(A,2) END LINE
```

First, displays the upper bound of **A**'s first subscript, then displays the upper bound of **A**'s second subscript.

```
    2   4
```

```
UBND(B,1);UBND(B,2) END LINE
```

First, displays the upper bound of **B**'s first subscript, then attempts to display the upper bound of **B**'s second subscript. Since **B** has only one subscript, `UBND(B,2)` returns −1.

```
 3  -1
```

```
LBND(A,1) END LINE
```

Displays the `OPTION BASE` setting when **A** was last dimensioned.

```
 1
```

Section 9

Inverse, Transpose and System Solution

Operations

INV

Matrix Inverse

```
MAT A=INV(B)
```

where **A** is a matrix and **B** is a square matrix.
Array **B** may be either real or complex type.
If **B** is complex, then **A** must be complex.
If **B** is real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as **B** and assigns to **A** the value of the matrix inverse of **B**.

To halt operation, press ATTN twice.

Not usable in CALC mode.

TRN

Matrix Transpose or Matrix Conjugate Transpose

```
MAT A=TRN(B)
```

where **A** and **B** are matrices.
Array **B** may be either real or complex type.
If **B** is complex, then **A** must be complex.
If **B** is real, then **A** may be real or complex; if complex, all imaginary parts of all elements in **A** are set to zero.

Implicitly redimensions **A** to be the same size as the matrix transpose of **B**. If **B** is real, assigns to **A** the value of the matrix transpose of **B**. If **B** is complex, assigns to **A** the values of the matrix conjugate transpose of **B**.

To halt operation, press ATTN twice.

Not usable in CALC mode.

Solving a System of Equations

The Math Pac provides a quick and accurate way to solve a system of linear equations involving real or complex coefficients. The first step in using this capability is to translate the system of equations into a triple of arrays: the result array, the coefficient array, and the constant array. The *result array* corresponds to the variables in the equations; the *coefficient array* holds the values of the coefficients of the variables; the *constant array* holds the values of the constants in the equations. For example, if you wanted to solve the system of equations

$$5x + 3y + 2z = 4$$

$$7x + y + 3z = 14$$

$$6x + 4y + 9z = 1$$

then the result array would correspond to the array

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

the coefficient array would be

$$\begin{bmatrix} 5 & 3 & 2 \\ 7 & 1 & 3 \\ 6 & 4 & 9 \end{bmatrix}$$

and the constant array would be

$$\begin{bmatrix} 4 \\ 14 \\ 1 \end{bmatrix}$$

If we denote the result array by **X**, the coefficient array by **A**, and the constant array by **B**, then the system of equations can be written in matrix notation as **AX=B**. This is the form assumed by the SYS keyword.

SYS

System Solution

MAT ■=SYS(**A**,**B**)

where **A** is a square matrix, **X** and **B** are both vectors or both matrices, and ■ and **B** are conformable for multiplication. Refer to the beginning of section 7, page 63, for a definition of "conformable for multiplication."

Arrays ■ and **B** may be either real or complex type.

If either **A** or **B** is complex, then ■ must be complex.

If both **A** and **B** are real, then **X** may be real or complex; if complex, all imaginary parts of all elements in **X** are set to zero.

Implicitly redimensions **X** to be the same size as **B** and assigns to **X** the computed solution to the matrix equation **AX=B**.

To halt operation, press ATTN twice.

Not usable in CALC mode

Examples

INV, TRN

Input/Result

```
OPTION BASE 1 END LINE
DIM A(3,3) END LINE
MAT A=IDN END LINE
MAT A=(2)*A END LINE
```

Assigns 2 to all diagonal elements of **A**. All other elements are zero.

```
MAT A=INV(A) END LINE
MAT DISP A; END LINE
```

Displays the inverse of **A**.

```
 .5  0  0
 0  .5  0
 0  0  .5
```

```
DIM C(3,2) END LINE
MAT C=CON END LINE
```

Assigns one to all elements of **C**.

```
MAT DISP C; END LINE
```

Displays **C**

```
 1  1
 1  1
 1  1
```

```
DIM D(2,2) END LINE
MAT D=TRN(C) END LINE
```

Computes the transpose of **C** and redimensions **D** to be a 2 × 3 matrix.

```
MAT DISP D; END LINE
```

Displays the transpose of **C**.

```
 1  1  1
 1  1  1
```

```
COMPLEX SHORT D(2,3),C(3,3)
END LINE
MAT D=(1,2) END LINE
```

Assigns the complex value (1,2) to all elements of **D**.

```
MAT DISP D; END LINE
```

```
 (1,2)  (1,2)  (1,2)
 (1,2)  (1,2)  (1,2)
```

The complex matrix **D**.

```
MAT D=TRN(D) END LINE
```

Redimensions **D** to 3 × 2 and assigns **D** the value of its conjugate transpose.

```
MAT DISP D; END LINE
```

```
 (1,-2)  (1,-2)
 (1,-2)  (1,-2)
 (1,-2)  (1,-2)
```

The conjugate transpose of **D**.

```
MAT INPUT C END LINE
```

```
C(1,1)? ■
```

```
1,(1,2),(2,10) END LINE
```

```
C(2,1)? ■
```

```
(1,1),(0,3),(-5,14) END LINE
```

```
C(3,1)? ■
```

```
(1,1),(0,5),(-8,20) END LINE
MAT DISP C; END LINE
```

```
 (1,0)   (1,2)   (2,10)
 (1,1)   (0,3)
 (-5,14)
 (1,1)   (0,5)
 (-8,20)
```

The complex matrix **C**.

```
MAT D=INV(C) END LINE
```

Redimensions **D** to 3 × 3 and assigns to **D** the value of the matrix inverse of **C**.

```
MAT DISP D; END LINE
```

```
 (10,1)   (-2,6)
 (-3,-2)
 (9,-3)
 (-7.09E-11,8)
 (-3,-2)
 (-2,2)   (-1,-2)
 (1,-1.1032E-11)
```

The inverse of the complex matrix **C** is the matrix

$$\begin{bmatrix} 10+i & -2+6i & -3-2i \\ 9-3i & 8i & -3-2i \\ -2+2i & -1-2i & 1 \end{bmatrix}$$

SYS

To solve the system of equations given on page 78, namely,

$$5x + 3y + 2z = 4$$

$$7x + y + 3z = 14$$

$$6x + 4y + 9z = 1$$

we could use the following keystrokes.

Input/Result

```
OPTION BASE 1 @ STD [END LINE]
DIM X(3),B(3),A(3,3) [END LINE]
MAT INPUT B,A [END LINE]
```

```
 B(1)? 
```

```
4,14,1 [END LINE]
```

Assigns values to the elements of **B**.

```
 A(1,1)? 
```

```
5,3,2,7,1,3,6,4,9 [END LINE]
MAT X=SYS(A,B) [END LINE]
MAT DISP X [END LINE]
```

Assigns values to the elements of **A**.

Displays the values of the result array elements.

```
  2.556603777358
 -2.650943396226
 -.4150943396226
```

$= x$.
$= y$.
$= z$.

Although in typical applications the result array **X** and constant array **B** are each one column arrays, SYS does not restrict these arrays to only one column. This allows you, for example, to simultaneously solve any number of different systems, limited only by memory, of n equations in n unknowns, provided that the coefficients in each systems of equations are identical. The following example illustrates this use of SYS.

Example. Your company's Publications Manager wants to determine the cost factors used by her two outside printers. She knows that each printer estimates jobs based on the number of pages and the number of photographs, plus a fixed setup charge. Given the three estimates from each printer shown below, write a program that calculates their cost per page, cost per photograph, and setup charge.

Job	Number of Pages	Number of Photographs	Total Cost	
			Printer A	Printer B
1	273	35	$5835.00	$7362.50
2	150	8	$3240.00	$4085.00
3	124	19	$2775.00	$3517.50

We need to solve the following system of equations for two sets of cost estimates.

$$273x_1 + 35x_2 + x_3 = cost_1$$

$$150x_1 + 8x_2 + x_3 = cost_2$$

$$124x_1 + 19x_2 + x_3 = cost_3$$

These equations can be represented in matrix notation as **AX** = **B**, where:

- **A** is the coefficient matrix, having the number of pages in its first column, the number of photographs in its second column, and the number of setup charges (one for each job) in its third column. Each row contains this data for a different job.
- ■ is the constant array. Each row contains cost estimates for one job from the two printers; each column contains one printer's cost estimates for the three jobs.
- **X** is the result array, having the unknown cost factors x_1, x_2, and x_3 in its rows. x_1 is the cost per page, x_2 is the cost per photograph, and x_3 is the setup charge. Since we are solving two systems, the constant array is ■ two-column matrix. So the result array must also be a matrix; that is, it should be declared with two dimensions. (Its size, if not the same size as that of the constant array **B**, will automatically be redimensioned to the size of **B** when the SYS statement is executed). Each column will contain the cost factors for one printer.

```
10 OPTION BASE 1
20 DIM A(3,3),X(3,2),B(3,2)
30 DATA 273,35,1
40 DATA 150,8,1
50 DATA 124,19,1
60 DATA 5835,7362.5
70 DATA 3240,4085
80 DATA 2775,3517.5
90 READ A,B
100 MAT X=SYS(A,B)
110 DISP USING '9A,3X,9A,/';
    'PRINTER A','PRINTER B'
120 MAT DISP USING 'X3D.2D,6X,
    3D.2D';X
```

Specifications for job 1.
Specifications for job 2.
Specifications for job 3.
Estimates for job 1.
Estimates for job 2.
Estimates for job 3.

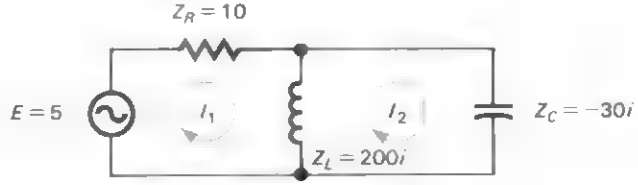
RUN

```
PRINTER A    PRINTER B

  20.00        25.00
   5.00         7.50
 200.00       275.00
```

Cost per page.
Cost per photograph.
Setup charge.

Example. This example demonstrates the usefulness of SYS in the solution of circuit analysis problems. In the circuit shown below, the impedances of the components are indicated in complex form. We will determine the complex representation of the currents I_1 and I_2.



This system can be represented by the complex matrix equation

$$\begin{bmatrix} 10+200i & -200i \\ -200i & (200-30)i \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

or

$$\mathbf{AX = B}$$

Here is a program that solves for I_1 and I_2.

```
10 OPTION BASE 1 @ STD
20 COMPLEX SHORT A(2,2),X(2)
30 DIM B(2)
40 MAT INPUT A,B
50 MAT X=SYS(A,B)
60 MAT DISP X
```

If either **A** or **B** is complex, **X** must be complex.

RUN

```
   A(1,1)? ■
```

```
(10,200),(0,-200),(0,-200),
(0,170) END LINE
```

Assigns values to the elements of **A**.

```
   B(1)? ■
```

```
5,0 END LINE
```

Assigns values to the elements of **B**.

```
    (.037156,.13114)
    (.043713,.15428)
```

I_1.
I_2.

Additional Information

The Math Pac operations DET(**A**), MAT **B**=INV(**A**), and MAT **B**=SYS(**A**,**B**), where **A** is a real-type square matrix, all use the *LU* decomposition of **A** as an intermediary step. The method used to generate the *LU* decomposition of **A** is a compact Crout factorization with partial pivoting and extended precision arithmetic. The *LU* decomposition of **A** can be represented by the equation **PA** = **LU**, where

- **L** is a lower triangular matrix—it has values of zero for all elements above the diagonal.
- **U** is an upper triangular martix—it has values of zero for all elements below the diagonal—with values of one for all elements on the diagonal.
- **P** is a permutation matrix representing the row interchanges in the matrix **A** resulting from partial pivoting.

The factorization **PA** = **LU** is valid for any non-singular matrix **A**. However, special attention is paid to matrices **A** that are singular or "machine singular." In this case, the *LU* decomposition is changed by an amount that is usually small in comparison with roundoff error. In the absence of underflow or overflow, the resulting *LU* decomposition of **A** will be close, in norm, to the exact *LU* decomposition of another matrix **A'**, where **A'** is close in norm to **A**.

Consider the matrix shown below.

$$\begin{bmatrix} 1 & 3 & 0 \\ 0 & 0 & 1 \\ .666666666667 & 2 & 0 \end{bmatrix}$$

Although this matrix is very nearly singular, it can be successfully inverted using the INV keyword:

Input/Result

```
OPTION BASE 1 [END LINE]
DIM A(3,3),B(3,3) [END LINE]
MAT INPUT A [END LINE]
```

```
A(1,1)? 
```

```
1,3,0,0,0,1 [END LINE]
```

```
A(3,1)? 
```

```
.666666666667,2,0 END LINE
```
A now represents the matrix given above.

```
MAT B=INV(A) END LINE
```
B is now the computed inverse of **A**.

```
MAT B=B*A END LINE
```

```
MAT DISP B; END LINE
```
Displays the identity matrix **B**, which is the product of the matrix **A** and its computed inverse.

```
 1   0   0
 0   1   0
 0   0   1
```

The SYS keyword solves the matrix equation **AX** = **B** for **X** in several stages. First, the *LU* decomposition of **A** is found to give **PA** = **LU**.

Using **PA** = **LU**, the equivalent problem is to solve **LUX** = **PB** for **X**. This is done by solving **LY** = **PB** for **Y** (*forward substitution*) and then solving **UX** = **Y** for **X** (*backward substitution*). This value for **X** is used as a first approximation to the desired solution in a process of iterative refinement, which produces the final result.

In many cases, the Math Pac will arrive at a correct solution even if the coefficient array is singular (so that the formula **X** = **A** $^{-1}$**B** is invalid). This feature allows you to use SYS to solve under- and over-determined systems of equations.

For an under-determined system (more variables than equations), the coefficient array will have fewer rows than columns. To find a solution using SYS:

- Append enough rows of zeros to the bottom of your coefficient array to make it square.
- Append corresponding rows of zeros to the constant array.

You can now use these arrays with the SYS keyword to find a solution to the original system.

For an overdetermined system (more equations than variables), the coefficient array will have fewer columns than rows. To find a solution using SYS:

- Append enough columns of zeros on the right of your coefficient array to make it square.
- Be sure that your result array is dimensioned to have at least as many rows as the new coefficient array has columns.
- Add enough zeros on the bottom of your constant array to ensure conformability.

You can now use these arrays with the SYS keyword to find a solution to the original system. Only those elements in the result array that correspond to your original variables will be meaningful.

For both under- and overdeterminded systems the coefficient array is singular, so you should check the results returned by SYS to see if they satisfy the original equation.

If **A** is a complex type square matrix, then MAT **C**=INV(**A**) and MAT **X**=SYS(**A**,**B**) use the same techniques as above, with the arrays **A** and **B** replaced by equivalent real-type partitioned forms.

The SYS keyword can also be used for inverting a square matrix **A**. MAT **X**=SYS(**A**,**B**) will return the inverse of **A** if **B** is chosen to be the identity matrix. This technique is more accurate and generally faster than MAT **X**=INV(**A**), but it requires more memory for its operation. (Refer to appendix B for information about memory requirements).

Section 10

Solving $f(x) = 0$

Keywords

You can use the keywords in this section to help you determine the solutions or minima of equations of from one to five real variables.

Throughout most of this section, the operation of these keywords will be described for a one-variable function. Multi-variable functions are covered under the topic *Nesting Rules.*

The keyword FNROOT can be used from the keyboard or inside a program to find the value of x for which $f(x)$ is zero or a minimum, provided the keyboard line or program contains the definition of the function.

The keywords FVALUE and FGUESS are provided to help you use FNROOT and to interpret its results. Since all three keywords are numeric-valued functions, they can be used alone or in combination with other functions and variables to form numeric expressions. A fourth keyword, FVAR, represents the variable in the function being solved by FNROOT. It also contains the most recent guess generated by an executing FNROOT.

FNROOT

Function Root

FNROOT(*A*,*B*,*F*)

where *A*, *B*, and *F* are real numeric expressions.

Seeks a real root of the function *F*, starting with the two guesses *A* and *B*. These guesses can be equal, but if so, one is immediately perturbed.

Returns the first value found that is one of the following:

1. An exact root of the specified function.
2. An approximation to a root of the specified function, correct to 12 digits.
3. An approximation to a local minimum of the absolute value of the specified function.
4. In ■ region where the specified function is constant.
5. ±9.99999999999E499 if the search for ■ root led beyond the range of representable numbers.

FNROOT (continued)

Not usable in CALC mode. Refer to page 97 for more information about FNROOT and CALC mode.

Refer to pages 97-99 for information about FNROOT nesting and about the interactions between FNROOT and [ATTN] and between FNROOT and user-defined functions.

FVAR

Function Variable

FVAR

Represents the variable *x* in *f*(*x*), the variable whose value FNROOT seeks.

Also returns the most current guess generated by a running FNROOT.

Can be used in CALC mode.

FVALUE

Function Value

FVALUE

Returns the value of the function *F* (the third argument of FNROOT) at the result generated by the most recently completed FNROOT.

FVALUE retains its value, even if your HP-71 is turned off, until FNROOT is again completed.

Can be used in CALC mode.

FGUESS

Previous Estimate of Function Root

FGUESS

Returns the next-to-last value tried as a solution in the most recently completed FNROOT statement.

FGUESS retains its value, even if your HP-71 is turned off, until FNROOT is again executed.

Can be used in CALC mode.

By checking the values of FVALUE and FGUESS, you can interpret the result of FNROOT as follows:

- If FVALUE = 0, the result of FNROOT is an exact root and the result of FGUESS will be a number close to the root.
- If the result of FNROOT and the result of FGUESS differ only in the twelfth significant digit, and FVALUE and F(FGUESS) have opposite signs, these two numbers surround the exact root.
- If the result of FNROOT and the result of FGUESS differ, but FVALUE and the value of the function at FGUESS are equal, these results lie in a region where FNF is constant.

To solve an equation for a particular variable, use this procedure:

1. Write the equation to be solved in the form $f(x) = 0$.
2. Substitute the keyword FVAR everywhere for the variable you wish to solve for in the formula defining $f(x)$.
3. Use the defining formula for $f(x)$ as the third argument for FNROOT.
4. Choose two initial guesses (which may be equal) and use these as the first two arguments for FNROOT. Even if only one initial guess is used, use it for both A and B, since FNROOT always requires three arguments.

Examples

Solving $x^2 = 2$ (FNROOT, FVALUE, FVAR)

The following six examples illustrate various ways FNROOT and FVAR can be used to solve the equation $x^2 = 2$. Initial guesses of 1 and 2 are used. The first and sixth examples show the solution.

Example One.

Input/Result

```
FNROOT(1,2,FVAR^2-2) END LINE
```

FNROOT can be used from the keyboard as well as in a program.

```
 1.41421356238
```

Example Two.

```
10 DISP FNROOT(COS(0),LOG2(4),
   FVAR^2-2)
20 DISP 'FVALUE =';FVALUE
```

The initial guesses can be expressions.

Example Three.

```
10 DEF FNG=FVAR^2-2
20 DISP FNROOT(1,2,FNG)

30 DISP 'FVALUE=';FVALUE
```

The third argument of FNROOT can be an expression or a reference to a user-defined function.

Example Four.

```
10 DEF FNF(X)=X^2-2
20 DISP FNROOT(1,2,FNF(FVAR))

30 DISP 'FVALUE=';FVALUE
```

FVAR can appear in the user-defined function, as above, or in the third argument of FNROOT.

Example Five.

```
10 DEF FNH

20 FNH=FVAR^2-2
30 END DEF
40 DISP FNROOT(1,2,FNH)
50 DISP 'FVALUE=';FVALUE
```

The user-defined function can consist of one or several lines.

Example Six.

```
10 DEF FNJ(X)
20 FNJ=X^2-2
30 END DEF
40 DEF FNF(X)=2*X
50 DISP FNROOT(1,FNF(1),FNJ(FVAR))

60 DISP 'FVALUE =';FVALUE
```

The first or second arguments of FNROOT can be references to user-defined functions.

Input/Result

RUN

```
 1.41421356238
FVALUE =  .000000000002
```

The solution for $x^2 = 2$.

Solving log (x) = e/x (FNROOT, FVALUE, FVAR, FGUESS)

To solve $\log(x) = e/x$, we first write the equation in the form $f(x) = 0$. This can be done by subtracting e/x from both sides of the equation, yielding $\log(x) - e/x = 0$. We can rewrite this in the equivalent but slightly more convenient form $x \log(x) - e = 0$. Since the left-hand side of this equation is undefined for $x <= 0$, and we can't guarantee that the search for a root will not venture into this region, we will consider instead the equation $|x| \log|x| - e = 0$. This equation has exactly the same positive solution(s) as the first equation, but this equation makes sense for both positive and negative (but nonzero) numbers. The program below includes a user-defined function that computes the left-hand side of this equation, and uses FNROOT to find a solution of the equation.

```
10 STD
20 DEF FNF(X)                          This user-defined function computes the left-
                                       hand side of the equation.
30 FNF=ABS(X)*LOG(ABS(X))-EXP(1)
40 END DEF
50 INPUT A,B                           These will be the initial guesses.
60 R=FNROOT(A,B,FNF(FVAR))
70 DISP 'R =';R
80 DISP 'FNF(R) =';FVALUE
90 DISP 'FGUESS=';FGUESS
```

To use the program we must decide on initial guesses. Although the initial guesses need not be in increasing order, or even distinct, a choice of initial guesses that surround a root will produce results more quickly in general. Noting that if $|FVAR| < 1$ then FNF(FVAR) will be negative and if FVAR is large (say, 100) then FNF(FVAR) will be positive, we can choose .5 and 100 for our initial guesses.

Key in the program and RUN it, and when prompted with ? respond with .5,100 END LINE, which supplies the initial guesses. The computer will then display

```
R = 2.71828182846
FNF(R) = 0
FGUESS= 2.76000738029
```

Since FNF(R) = 0, the value given is an exact root for FNF.

Additional Information

Choosing Initial Estimates

When you use FNROOT to find roots of equations, the initial estimates determine where the search for a root will begin. If the two estimates surround an odd number of roots (signified by their function values having opposite signs), then FNROOT will find a root between the estimates quite rapidly. If the function values at the two estimates do not differ in sign, then FNROOT must search for a region where a root lies. Selecting initial estimates as near a root as possible will tend to speed up this search. If you merely want to explore the behavior of the function near the initial etimates (such as to determine if there are any roots or extreme points nearby), then specify any estimates you like.

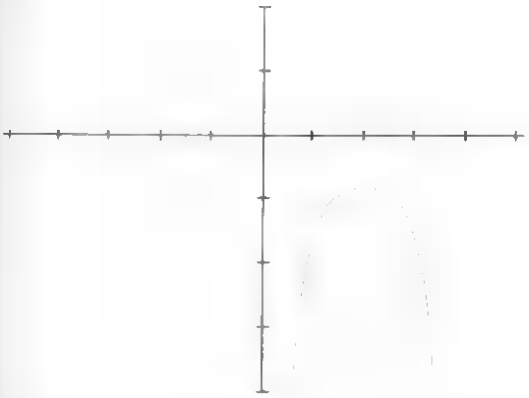
Another thing to consider is the range in which the equation is meaningful. In solving $f(x) = 0$, the variable x may only have a limited range in which it is conceptually meaningful as a solution. In this case, it is reasonable to choose initial estimates within this range. Frequently an equation that is applicable to a real problem has, in addition to the desired solution, other roots that are physically meaningless. These usually occur because the equation being analyzed is appropriate only between certain limits of the variable. You should recognize this restriction and interpret the results accordingly.

Interpreting Results

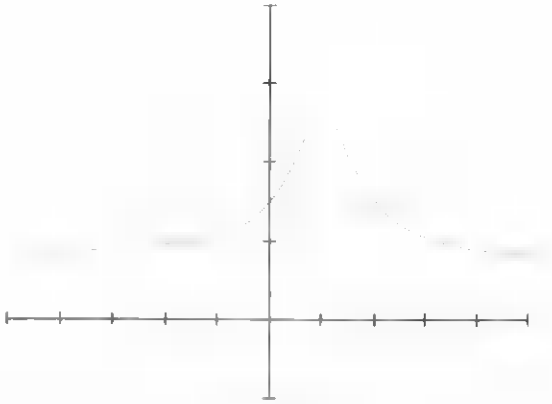
FNROOT always evaluates the function at the value returned, as described above. This enables you to interpret the results. There are two possibilities: the value of the function at the value returned by FNROOT in close to 0; or the value of the function at the value returned by FNROOT is not close to 0. It is up to you to decide how close is close enough to consider the value a root.

If the function value is too large, then the information returned by the keyword FGUESS, together with information already considered, is sufficient to determine the general behavior of the function in the region. For example, suppose that FNROOT is used to find a root of a function—say, $f(x)$—and the value returned is r. If |FVALUE| is too large to consider r a root, then there are several possibilities.

If FVALUE and f(FGUESS) have the same sign, then r is either an approximation to a local minimum of $|f(x)|$ or in a region where the graph of $f(x)$ is horizontal.



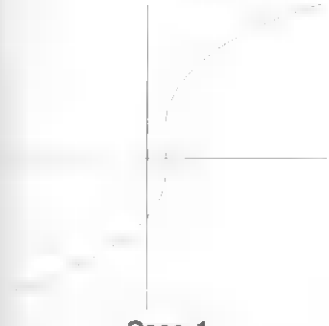
Case a



Case b

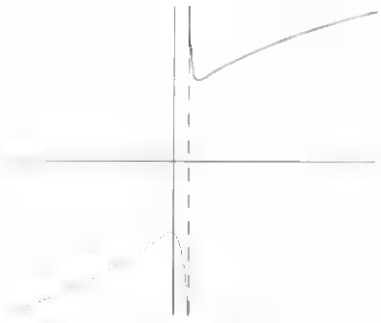
In the two cases above, FNROOT sees no tendency of $f(x)$ to decrease in absolute value, and so to cross the x-axis. It will then try to approximate a local extreme point, if any. This approximation can be resolved to further precision by further executions of FNROOT, using r and FGUESS as initial estimates. Repeated execution of FNROOT in this manner will converge to the extreme point in many cases. The idea is that FNROOT can be used to find local extreme points, or the information about where the extreme points are can be used to re-direct the search elsewhere, in hope of finding a root.

When |FVALUE| is too large to consider r a root, another possibility is that FVALUE and f(FGUESS) have different signs. In this case it would appear that there is a root between, because for the function to change signs it should cross the x-axis. Typically, when FNROOT finds two guesses on opposite sides of the x-axis, it only stops after it has resolved them to two consecutive machine numbers. In this case there is no machine representable number between r and FGUESS. Thus, the behavior of the function cannot be determined between r and FGUESS. To interpret such results, you should be aware of these situations.



Case 1

Case 2



Case 3

In case 1, r and FGUESS are the best approximations to the root that are representable on the machine. Case 2 looks exactly the same to FNROOT, but there is no root—there is a jump discontinuity instead. In case 3 there is a pole, which can look like a root if a guess on each side of the pole is found. FNROOT returns information in FGUESS and the FVALUE to help you isolate situations where convergence is to a pole.

Decreasing Execution Time

The exponent range of your HP-71 is ±499 (except when TRAP(UNF) = 2, which effectively extends the negative exponent range to −510). This allows for sensitive observation of the behavior of a function, even very close to a root. FNROOT takes advantage of this dynamic range by not accepting a guess as a root until the function value underflows, is zero, or two consecutive machine representable numbers that bracket a root are found. The cost of this precision is that, occasionally, it may take quite a while to obtain such precision. If this high degree of sensitivity is not required, then you may wish to set a smaller tolerance. For example, you may only need to know a place where the function is less than 1E−20. This is accomplished in your function definition by checking the value of the function before assigning it to the function variable and setting the function variable to zero if the computed value is smaller than the desired tolerance. For example, suppose you wanted to find any root of $f(x) = x^4$, and $|f(x)| <= 1E-32$ is acceptable as a root. Here is a program you can use.

```
10 STD
20 DEF FNF(X)                          Multiline function definition of f(x) = x^4.
30 F=X^4
40 IF F<=1.E-32 THEN FNF=0 ELSE        Checks error tolerance and sets the function
   FNF=F                               value accordingly.
50 END DEF
60 DISP FNROOT(2,3,FNF(FVAR))          Computes and displays the root.
70 DISP FVALUE                         Displays the function value at the root.
```

Input/Result

RUN

```
 9.30442502653E-9
 0
```

In this example, if the *tolerance technique* were not used, execution would last much longer. This is because the computed function will not underflow until x is very small, since the root is at zero and the distribution of machine-representable numbers is very dense close to zero. So FNROOT has a lot of guesses to try before finding one it can accept as a root.

An alternate approach to decreasing execution time is to translate the function so that the root is not so near zero, compute the root of the translated function, then translate the root back. This will decrease the time to find roots of certain functions with roots close to zero, but will generally decrease the accuracy of the roots found. Here is a sample program for $f(x) = x^4$.

```
10 STD
20 DEF FNF(X)=(X-1)^4          This is x^4 translated by 1.
30 R=FNROOT(3,4,FNF(FVAR))     Computes the root.
40 DISP R-1                    Translates the root back and displays the root
                               and function value.
50 DISP FVALUE
```

Finally, there is a technique that may improve the speed and accuracy of FNROOT. Any equation is typically one of an infinite family of equivalent equations with the same roots. However, some may be easier to solve than others. For example, the two equations $f(x) = 0$ and $\exp(f(x)) - 1 = 0$ have the same real roots, but one is almost always easier to solve. When $f(x) = x^4 - 6x - 1$, the first is easier; but when $f(x) = \ln(x^4 - 6x - 1)$, then the second is easier. While FNROOT has been designed to provide accurate results for a wide range of problems, it is worthwhile to be aware of such possibilities.

Suspending FNROOT With ATTN

If none of the arguments of FNROOT contain multi-line user-defined function calls, pressing ATTN will not terminate the action of FNROOT until intermediate results are saved. In particular, FNROOT will return and save the current FVAR as though it were a root, it will save the previous guess as FGUESS, and it will save the value of $f(x)$ at the current FVAR as the value of FVALUE. Only then will the action of FNROOT stop.

If, on the other hand, there are one or more multi-line user-defined functions as arguments for FNROOT (that is, if the calculation of FNROOT involves several BASIC program lines), ATTN will be ignored until a multi-line user-defined function is called. Execution will then halt at a line of one of the user-defined functions. This gives you the ability to examine relevent values, such as the current value of FVAR, then continue the execution of FNROOT if you wish.

In addition, if there are multi-line user-defined functions as arguments for FNROOT, then fatal errors within the user-defined function do not destroy the FNROOT environment, giving you the exact same correct and continue capability as with any other HP-71 call to multi-line user-defined functions.

CALC Mode

You cannot execute FNROOT directly or indirectly in CALC mode. For instance, suppose your current file contains a single-line user-defined function FNF whose definition contains an FNROOT keyword. If you attempt to execute FNF in CALC mode, an error will result.

Nesting Rules

If the third argument F of FNROOT defines a formula whose evaluation encounters another FNROOT keyword, we say that the two FNROOT keywords are nested. Up to five FNROOT keywords can be nested in this way.

As an example of FNROOT nesting, consider the following program that solves $f(x,y) = x^2 + y^2 - 2x - 2y + 2$ for x and y.

```
10 STD
20 DEF FNF(X,Y)=X^2+Y^2-2*X-2*Y+2
30 DEF FNG(X)
40 R=FNROOT(-4,4,FNF(X,FVAR))
50 FNG=FVALUE
60 END DEF
70 DISP FNROOT(-3,3,FNG(FVAR));R
```

Defines the function whose solution is sought.

Lines 30 through 60 define a one variable function $f(x)$ that receives a fixed x value (FVAR) from line 70.

If this FNROOT function receives a nonzero result from line 50, it selects another x value for the FNROOT in line 40 to try. If it receives a zero result, a solution for $f(x,y)$ is found.

Input/Result

RUN

```
 1    .999999999999
```

The x and y values returned by the FNROOT function in line 70. The x value is displayed on the left.

The closest FNROOT comes to the true y value, one, is .999999999999, since these x and y values satisfy the objective of FNROOT. This objective is to find x and y values for which the computed value of $f(x,y)$ is zero.

FVALUE END LINE

```
 0
```

These values for x and y when used in $f(x,y)$ give 0 as the result.

A common use for FNROOT nesting is determining minima. To demonstrate this application, we'll modify the above function $f(x,y)$ by adding one to the expression, thereby ensuring that the function has no solution, since the paraboloid represented by the modified function no longer intersects the xy plane. The only program modification is in line 20:

```
20 DEF FNF(X,Y)=X^2+Y^2-2*X-2*Y+3
```

All other program lines are unchanged.

The earlier nested FNROOT program required about 20 seconds to reach a solution. Since FNROOT takes special care to make sure a true minimum is found, the modified program requires about 3½ minutes to find and display the x and y values whose use in $f(x,y)$ result in a function minimum.

Input/Result

RUN

```
 1.000000191832   1.0000
014444
```

The x and y values that give a minimum for the modified function.

FVALUE END LINE

Displays the value given by the modified function using these x and y values.

```
 1
```

There is no need to wait the full 3½ minutes for a result. As explained on page 97, you can suspend an executing FNROOT function and then display interim results. If two consecutive inspections of interim results show insignificant change, you might wish to accept them as having satisfactory accuracy.

Use of User-Defined Functions

If the third argument of an FNROOT function evaluates any user-defined function, then you must execute the FNROOT function as a program statement, not from the keyboard. Also, if FNROOT is suspended while executing, you cannot execute a user-defined function from the keyboard, in either BASIC or CALC mode.

Section 11

Numerical Integration

Keywords

You can use the keywords in this section to evaluate the integral of a function of from one to five variables between definite limits to an accuracy of your choosing.

Throughout most of this section, the operation of these keywords will be described for a one-variable function. Multi-variable functions are covered under the topic *Nesting Rules - Volume Integration*, pages 109-110.

The keyword INTEGRAL can be used from the keyboard or inside a program to calculate the integral of the function, provided the keyboard line or program contains the function definition.

The keywords IBOUND and IVALUE give you additional flexibility in the evaluation of the integrals. INTEGRAL, IBOUND, and IVALUE are numeric-valued, so they can be used alone or in combination with other functions and variables to form numeric expressions. A fourth keyword, IVAR, represents the variable (or one of the variables) of integration in the function being integrated by INTEGRAL. It also contains the most recent sampling point used by an executing INTEGRAL.

INTEGRAL — Definite Integral

INTEGRAL(*A*,*B*,*E*,*F*)

where *A*, *B*, *E*, and *F* are real numeric expressions.

Returns an approximation to the integral from *A* to *B* of *F*. The relative error *E* (rounded to the range 1E-12<=*E*<=1) indicates the accuracy of *F* and is used to calculate the acceptable error in the approximation of the integral.

This integral approximation can be:

- An approximation to the integral that is as accurate as the relative error *E* would allow.
- The last of 16 approximations to the integral, which have sampled the integrand at 65535 points without meeting the convergence criterion.
- The best current approximation to the integral returned when ATTN is pressed and when *F* does not call ■ multi-line user-defined function.

INTEGRAL (continued)

INTEGRAL generates a sequence of increasingly accurate approximations to the definite integral. If three successive approximations are within the acceptable error of each other—the first is close to the second and the second is close to the third—they are considered to have converged and the third approximation is returned as the value of the definite integral. If a total of 16 approximations are calculated without converging, the sixteenth is returned.

Not usable in CALC mode. Refer to page 111 for more information about INTEGRAL and CALC mode.

Refer to pages 109-111 for information about INTEGRAL nesting (volume integration) and about the interactions between INTEGRAL and [ATTN] and between INTEGRAL and user-defined functions.

IVAR — Integration Variable

IVAR

Represents the variable of integration in the formula defining *F*, the last argument of INTEGRAL.

Also contains the most recent sampling point used by a running INTEGRAL.

Can be used in CALC mode.

IVALUE — Last Result of INTEGRAL

IVALUE

Returns the last approximation computed by the INTEGRAL keyword. If the [ATTN] key was pressed or the operation of INTEGRAL was otherwise interrupted, then IVALUE returns the value of the current approximation to the integral. Otherwise, IVALUE returns the same value that INTEGRAL last returned.

IVALUE retains its value (even if your HP-71 is turned off) until another INTEGRAL is computed.

Can be used in CALC mode.

IBOUND

Error Approximation for INTEGRAL

IBOUND

Returns the final *absolute* error estimate for the definite integral most recently computed by INTEGRAL.

- A positive value for IBOUND means that the approximations converged.
- A negative value for IBOUND means that the approximations didn't converge, so that the value returned by INTEGRAL may not be representative of the true value.

Like IVALUE, IBOUND retains its value (even if the HP-71 is turned off) until another INTEGRAL is computed. Unlike IVALUE, the value of IBOUND has no relation to the current approximation to the integral if the operation of INTEGRAL is interrupted.

Can be used in CALC mode.

To integrate a function between bounds, you can follow these steps:

1. Write down an expression that represents the function to be integrated.
2. Substitute the keyword IVAR everywhere in the expression for the variable of integration.
3. Use this expression as the fourth argument F of INTEGRAL.
4. Use the lower and upper bounds of integration as the first and second arguments A and B of INTEGRAL, respectively.
5. Choose a value for the third argument E of INTEGRAL that represents an estimate of the *relative* error in the computation of the integrand. Any value for E is always rounded to the range [1E−12,1]. Thus, E should satisfy, after rounding

$$\frac{|\text{TRUE INTEGRAND} - \text{COMPUTED INTEGRAND}|}{|\text{COMPUTED INTEGRAND}|} <= E.$$

Since INTEGRAL has no way of knowing what the true value of the function is intended to be, only you can supply this estimate. For many purely mathematical functions (SIN, EXP, polynomials, etc.) and modest limits of integration, full 12 digit accuracy can be returned so that a value for E around 1E-12 should be suitable.

The operation of INTEGRAL and IBOUND can be described more precisely as follows.

1. Based on a relative error of E for the specified function, the computer calculates an error tolerance for the integral of the specified function. If $f(X)$ is the "true" function that F approximates, then choose E such that

$$\frac{|F - f(X)|}{|F|} \leq E$$

for all X in the interval of integration. Your input for E is rounded to the range $1E-12 \leq E \leq 1$.

For example, if F is derived from experimental data with N significant digits, let E equal 10^{-N}.

2. The computer calculates a sequence of approximations I_k to the integral of the specified function. The difference between successive approximations is compared to the error tolerance for the integral.

3. A value for the integral is returned when

- The approximations I_k have converged. Convergence is determined using J_k, defined as the kth approximation to the integral of $E*|F|$ over the same interval of integration. J_k represents the error inherent in the computation of I_k.

  The approximations I_k are judged to have converged to I_n if

$$|I_k - I_{k-1}| \leq J_k$$

  for $k = n - 1$ and $k = n$. The value of I_n is then returned by INTEGRAL; a positive value for the error estimate will be returned by IBOUND.

  or when

- The computer has evaluated I_1 through I_{16} but the convergence criterion is still not met. I_{16} is then returned by INTEGRAL; a negative value for the error estimate will be returned by IBOUND.

Examples

Integrating $f(x) = x^2-2$ (INTEGRAL, IVAR)

The following six examples illustrate various ways INTEGRAL and IVAR can be used to integrate the function $x^2 - 2$ from 1 to 2. The first and sixth examples show the solution.

Example One.

Input/Result

```
INTEGRAL(1,2,1E-11,IVAR^2-2)
END LINE
```

```
 .333333333331
```

INTEGRAL can be used from the keyboard as well as in a program.

Example Two.

```
10 DISP INTEGRAL(COS(0),LOG2(4),
   1E-11,IVAR^2-2)
```

The limits of integration can be expressions.

Example Three.

```
10 DEF FNG=IVAR^2-2
20 DISP INTEGRAL(1,2,1E-11,FNG)
```

The fourth argument of INTEGRAL can be an expression or a reference to a user-defined function.

Example Four.

```
10 DEF FNF(X)=X^2-2
20 DISP INTEGRAL(1,2,1E-11,FNF(IVAR))
```

IVAR can appear in the user-defined function, as above, or in the fourth argument of INTEGRAL.

Example Five.

```
10 DEF FNH
20 FNH=IVAR^2-2
30 END DEF
40 DISP INTEGRAL(1,2,1E-11,FNH)
```

The user-defined function can consist of one or several lines.

Example Six.

```
10 DEF FNJ(X)
20 FNJ=X^2-2
30 END DEF
40 DEF FNF (X)=2*X
50 DISP INTEGRAL(1,FNF(1),1E-11,
     FNJ(IVAR))

60 DISP IBOUND
```

The first, second or third arguments of INTEGRAL can be references to user-defined functions.

Input/Result

RUN

```
      .333333333331
```

The resulting integral.

```
      7.70341735781E-12
```

The absolute error estimate for the resulting integral. Since it's positive, the approximations converged.

Integrating $f(x) = e^x - 2$ (INTEGRAL, IVAR, IVALUE)

This example features IVALUE. This function returns the most recent integration approximation and is updated even while the execution of INTEGRAL is in progress. After the execution of INTEGRAL is completed, IVALUE returns the same value returned by INTEGRAL.

You can watch the progress of integral approximations by displaying IVALUE during the execution of INTEGRAL. This is demonstrated by the following program, which integrates the function $e^x - 2$ from one to three. The error bound used is 1E-12.

```
10 Y=IVALUE
```

Y = value of IVALUE when program starts (assumes IVALUES is set from a previous INTEGRAL).

```
20 DEF FNF(X)
30 IF IVALUE=Y THEN 50
```

Displays IVALUE only if it has changed.

```
40 DISP IVALUE @ Y=IVALUE
50 FNF=EXP(X)-2
60 END DEF
70 DISP INTEGRAL(1,3,.000000000001,
    FNF(IVAR))
```

Input/Result

RUN

```
    10.7781121979
    13.683897213
    13.3653590516
    13.3671560314
    13.3672555263
    13.3672550945
    13.3672550947
    13.3672550947
```

First displayed value of IVALUE.

Last displayed value of IVALUE.
Value of INTEGRAL.

Integrating $f(x) = \exp(x^3+4x^2+x+1)$ (INTEGRAL, IVAR, IBOUND, IVALUE)

To find the integral from 0 to 1 of the function

$$f(x) = \exp(x^3 + 4x^2 + x + 1)$$

you can use the following program.

```
10 DEF FNF(X)=EXP(X^3+4*X^2+X+1)
20 INPUT E

30 DISP 'Integrating'
40 X=INTEGRAL(0,1,E,FNF(IVAR))
50 BEEP
60 DISP 'Integral =';X
70 DISP 'The approx. error ='
80 DISP IBOUND
```

The user-defined function FNF.

Gets the relative error we expect in FNF as compared with *f*.

After you key in the program, run it using the following keystrokes.

Input/Result

RUN

```
? ■
```

The prompt to enter the relative error of the function.

1E-5 END LINE

Although our function is accurate to one part in 10^{12}, we can say that it is less accurate (in this case, one part in 10^5) so that the computation will finish more quickly.

```
Integrating
```

```
Integral =
 104.291097226
The approx. error =
1.04263904392E-3
```

The value of the integral is 104.2911 $\pm$ (1.04 $\times$ 10^{-3}).

IVALUE END LINE

```
 104.291097226
```

IVALUE gives the value of the last computed integral.

Integrating $C(T) = a + bT$ (INTEGRAL, IVAR, IBOUND)

You can use INTEGRAL to compute the amount of heat required to heat one gram of gas at a constant volume from one temperature to another. The amount of heat needed, Q, is given by the formula

$$Q = \int_{T1}^{T2} C(T)\, dT$$

where $C(T)$ is the specific heat of the gas as a function of temperature, $T1$ is the starting temperature, and $T2$ is the final temperature.

If $C(T) = a + bT$, where a and b are experimentally determined to be $a = 1.023\text{E}^{-2}$ and $b = 2.384\text{E}^{-2}$ with four significant digits, then we can compute the relative error of $C(T)$ to be approximately 5E−4. The program below prompts you for the initial and final temperature in degrees Kelvin and then computes the heat needed to raise the temperature of the gas from the initial to the final temperature.

```
10 DEF FNC(T)=.01023+.02384*T
20 INPUT 'Initial, final T (K)?';T1,T2
30 DISP 'Integrating'
40 Q=INTEGRAL(T1,T2,.0005,FNC(IVAR))
50 DISP 'Heat needed =';Q;'+-';IBOUND
```

The user-defined function that calculates the specific heat.

Computes the integral.

Displays the answer and the approximate error.

To find the heat needed to raise the temperature from 300°K to 310°K, type in the program and use the following keystrokes.

Input/Result

RUN

```
Initial, final T (K)?
```

300,310 END LINE

```
Integrating
Heat needed = 72.8143
+- .03640715
```

Additional Information

Nesting Rules—Volume Integration

If the fourth argument *F* of INTEGRAL defines a formula whose evaluation encounters another INTEGRAL keyword, we say the two INTEGRAL keywords are nested. Up to five INTEGRAL keywords can be nested in this way. A program that nests two INTEGRAL keywords can determine volumes.

As an example of INTEGRAL nesting, consider the following program that integrates $f(x,y)$, where $f(x, y) = x^2 + 2y$, over the square $0 < x < 1$, $0 < y < 1$. That is, the program evaluates

$$\int_0^1 \int_0^1 f(x, y) dy dx.$$

```
10 DEF FNF(X,Y)=X^2+2*Y
20 DEF FNG(X)=INTEGRAL(0,1,1E-6,
     FNF(X,IVAR))
30 INTEGRAL(0,1,1E-6,FNG(IVAR))
```

Defines the function whose integral is sought.

For each value of X, integrates a slice parallel to the y-axis.

Sums all of the contributions from the slices parallel to the y-axis.

Input/Result

RUN

```
 1.33333333333
```

The volume returned by the INTEGRAL function in line 30.

IBOUND END LINE

```
 1.33317012712E-6
```

The answer is exact even though IBOUND only predicts six correct digits.

The following example demonstrates the use of INTEGRAL to evaluate the integral

$$\int_0^{\pi/2}\int_0^{y} \sin(x)\, dxdy$$

Input/Result

```
RADIANS END LINE
INTEGRAL(0,PI/2,1E-3,
INTEGRAL(0,IVAR,1E-3,
SIN(IVAR))) END LINE
```

Note that the first IVAR is the integration variable of the outside INTEGRAL.

```
 .57080016668
```

IBOUND END LINE

```
 5.69950328155E-4
```

The true answer is $\pi/2 - 1$ (approximately .5707963268).

Suspending INTEGRAL With [ATTN]

If none of the arguments of INTEGRAL contain multi-line user-defined function calls, pressing [ATTN] will not terminate the action of INTEGRAL until intermediate values are saved. In particular, INTEGRAL will save and return the current IVALUE as though it were the integral, and will make negative the current value of IBOUND. Only then will the action of INTEGRAL stop.

If, on the other hand, there are one or more multi-line user-defined functions as arguments for INTEGRAL (that is, if the calculation of INTEGRAL involves several BASIC program lines), [ATTN] will be ignored until a multi-line user-defined function is called. Execution will then halt at a line of one of the user-defined functions. This gives you the ability to examine relevent values, such as the current value of IVALUE, then continue the execution of INTEGRAL if you wish.

In addition, if there are multi-line user-defined functions as arguments for INTEGRAL, then fatal errors within the user-defined function do not destroy the INTEGRAL environment, giving you the exact same correct and continue capability as with any other HP-71 call to multi-line user-defined functions.

CALC Mode

You cannot execute INTEGRAL directly or indirectly in CALC mode. For instance, suppose your current file contains a single-line user-defined function FNF whose definition contains an INTEGRAL keyword. If you attempt to execute FNF from CALC mode, an error will result.

Use of User-Defined Functions

If the fourth argument of an INTEGRAL function evaluates any user-defined function, then you must execute the INTEGRAL function as a program statement, not from the keyboard. Also, if INTEGRAL is suspended while executing, you cannot execute a user-defined function from the keyboard, in either BASIC or CALC mode.

Overview of Numerical Integration

Numerical integration schemes generally sample the function to be integrated at a number of points in the interval of integration. The calculated integral is simply a weighted average of the function values at these sample points. Since a definite integral is really an average value of a function over an *infinite* number of points, numerical integration can produce accurate results only when the points sampled are truly representative of the function's behavior.

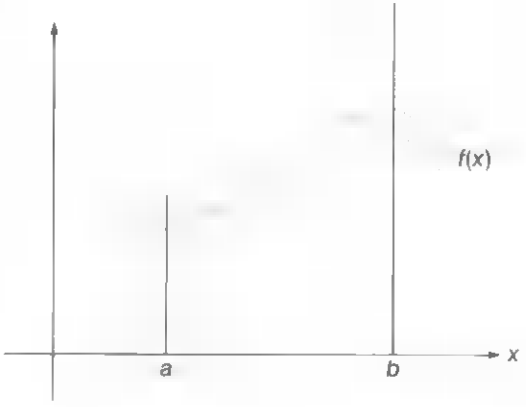
If the sample points are close together and the function does not change rapidly between two consecutive sample points, then the numerical integration will give reliable results. On the other hand, numerical integration will not produce good answers on a function whose values vary wildly over a domain that is small in comparison with the region of integration. Other errors that can affect the result of a numerical integration include the round-off errors typical of any floating point computation and errors in the procedure that computes the function to be integrated.

Handling Numerical Error

The INTEGRAL keyword requires specification of an error tolerance E for its operation. This error tolerance is taken to be the *relative error* of the computed function as compared with the "true" function to be integrated. The error tolerance is used to define a ribbon around the computed function and the "true" function should then lie inside this ribbon. If the "true" function is $f(x)$ and the computed function is $F(x)$, then

$$F(x) - \text{Error } (x) \leq f(x) \leq F(x) + \text{Error } (x)$$

where Error (x) is half the width of the ribbon at x.



We can then conclude that

$$\int_a^b f(x)\,dx \approx \int_a^b F(x)\,dx \pm \int_a^b \text{Error } (x)\,dx$$

where the third integral is just half the area of the ribbon—that is, integrating the computed function instead of the actual function can introduce an error no greater than half the area of the ribbon. INTEGRAL estimates this error while computing the integral; IBOUND gives you access to the estimate.

Choosing the Error Tolerance

The accuracy of the computed function depends on three factors:

- The accuracy of empirical constants in the function.
- The degree to which the function may accurately describe a physical situation.
- The round-off error introduced when the function is computed.

Functions like $\cos(x - \sin x)$ are purely mathematical functions. This means that the functions contain no empirical constants, and neither the variables nor the limits of integration represent any actual physical quantities. For such functions you can specify as small an error tolerance as desired, provided the function is calculated within that error tolerance (despite round-off) by the BASIC function. Of course, due to the trade-off between accuracy and computation time, you may choose not to specify the smallest possible error tolerance. Any specified error tolerance is rounded to the range [1E−12, 1].

When the integrand relates to an actual physical situation, there are additional considerations. In these cases, you must ask yourself whether the accuracy you would like in the computed integral is justified by the accuracy of the integrand. For example, if the function contains empirical constants that approximate the actual constants to three digits, then it may not make sense to specify an error tolerance smaller than 1E−3.

An equally important consideration, however, is that nearly every function relating to a physical situation is inherently inaccurate because it is only a mathematical model of an actual process or event. A mathematical model is typically an approximation that ignores the effects of factors judged to be insignificant in comparison with the factors in the model.

For example, the equation $s = s' - (.5)gt^2$, which gives the height s of a falling body when dropped from an initial height s', ignores the variation with altitude of g, the acceleration due to gravity. Mathematical descriptions of the physical world can provide results of only limited accuracy. If you calculated an integral with an accuracy greater than your model can support, then you would not be justified in using the calculated value to its full (apparent) accuracy. It makes sense to supply an error tolerance that reflects any inaccuracies in the function, or the INTEGRAL keyword will waste time computing to a level of accuracy that may be meaningless. Further, the value returned by IBOUND may not be significant.

If $f(x)$ is a function relating to a physical situation, its inaccuracy due to round-off is typically very small compared to the inaccuracy in modelling the situation. If $f(x)$ is a purely mathematical function, then its accuracy is limited only by round-off error. Precisely determining the relative error in the computation of such a function generally requires a complicated analysis. In practice, its effects are determined through experience rather than analysis.

Handling Difficult Integrals

Integrating on Subintervals. A function whose values change substantially with small changes in its argument will likely require many more points than one whose values change only slightly. This is because the behavior of the function must be adequately represented by the sampling. If a function is changing more rapidly in some subintervals of the interval of integration than in others, you can subdivide the interval and integrate the function separately on the smaller intervals. Then the integral over the whole interval is the sum of the integrals over all the subintervals, and the error of the integral is the sum of the errors of the integrals over the subintervals.

The algorithm used by INTEGRAL makes a reasonable decision during execution of how many points to sample, based on the behavior of the specified integrand on a particular interval. When the interval of integration is split up, each subinterval can be handled according to the function behavior on that subinterval alone. This results in greater speed and precision.

For example, to integrate $f(x) = (x^2 + 1\text{E}{-}12)^{1/2}$ from $x = -3$ to $x = 5$ using an error tolerance of 1E−12, it speeds up execution to subdivide the interval at $x = 0$, where $f(x)$ has a sharp bend in its graph. Because $f(x)$ is very smooth on the subintervals $(-3, 0)$ and $(0, 5)$, the integrals over these subintervals can be evaluated quickly.

$$\int_{-3}^{5} f(x)\,dx = \int_{-3}^{0} f(x)\,dx + \int_{0}^{5} f(x)\,dx$$

The following program computes this integral on the two subintervals and then combines the results.

```
10 DEF FNF(X)=SQR(X*X+1E-12)
20 I=INTEGRAL(-3,0,1E-12,FNF(IVAR))
30 E=IBOUND
40 DISP "Integral =";
50 DISP I+INTEGRAL(0,5,1E-12,FNF(IVAR))
60 DISP "Error =";E + IBOUND
```

We will use X*X rather than X^2 because X*X is more accurate. An analogous situation generally occurs for any *integer* power of a variable.

Integrate over the first subinterval.

Save the error to add in later.

The sum of the first and second integrals.

Compute the relative error by adding the two errors together.

You can run this program by keying it in and then pressing RUN. The following will then appear in the display.

```
Integral = 17
Error =  .000000000017
```

When the interval is subdivided, INTEGRAL computes the answer in a few seconds. Without subdividing the interval, execution may take a long time.

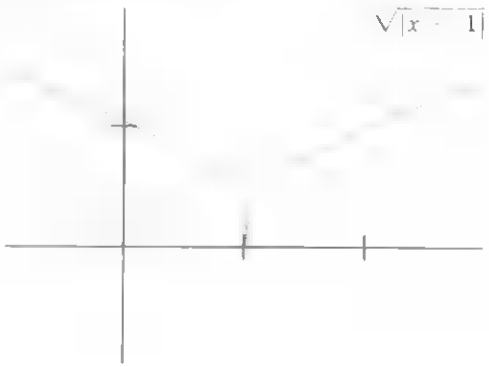
Subdividing the interval of integration is also useful for functions with a singularity in the interval. The singularity may consist of one or more points where the function is undefined or has a sharp corner point.

For example, the integral

$$\int_0^2 \frac{dx}{(x-1)^2} \quad \text{may be split into} \quad \int_0^1 \frac{dx}{(x-1)^2} + \int_1^2 \frac{dx}{(x-1)^2}$$

to avoid evaluating the function at $x = 1$, where it is undefined. You can now integrate the function on each subinterval because $x = 1$ is an endpoint of each subinterval, and INTEGRAL does not sample at an endpoint.

Similarly, the function $\sqrt{|x-1|}$, has a sharp corner point at $x = 1$.

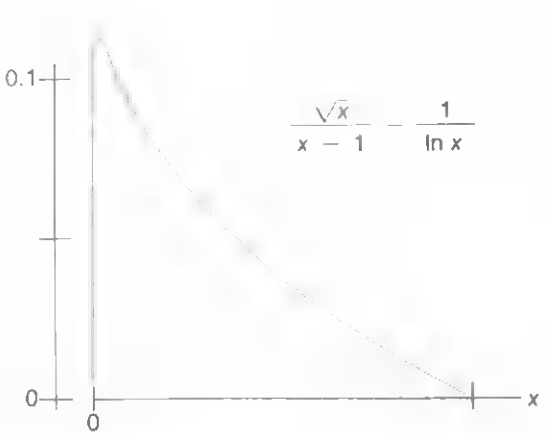


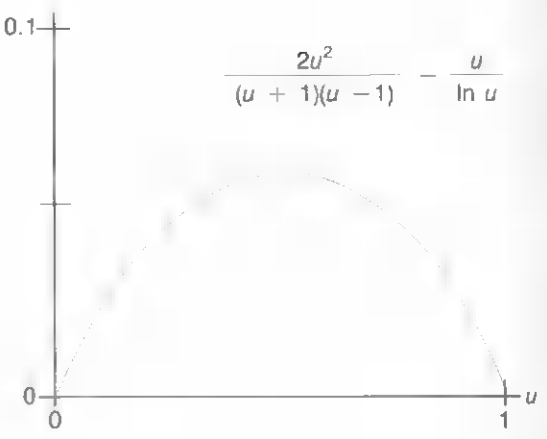
Suppose you need to integrate this function from 0 to 2. You can increase the speed and accuracy of the computation by integrating separately on the subintervals (0, 1) and (1, 2), because the function is smooth on each of these subintervals.

Transformation of Variables. A second method of handling difficult integrands is by transforming the variable. When the variable in a definite integral is transformed, the resulting definite integral may be easier to compute numerically. Consider the integral

$$\int_0^1 \left(\frac{\sqrt{x}}{x-1} - \frac{1}{\ln x} \right) dx.$$

The derivative of the integrand approaches infinity as x approaches 0, as shown on the left below. The substitution $x = u^2$ stretches the x-axis and causes the function to be better behaved, as shown on the right.





You can now evaluate the integral that results from this substitution:

$$\int_0^1 \left(\frac{2u^2}{(u+1)(u-1)} - \frac{u}{\ln u} \right) du.$$

(Do not replace $(u + 1)(u - 1)$ with $u^2 - 1$; as u approaches 1, $u^2 - 1$ loses half of its digits to roundoff, yielding a final result that is too large.)

As a second case requiring substitution, consider the following function. Its graph has a long tail stretching much farther than the main body (where most of the area is).

Although a very thin tail may be truncated without greatly degrading accuracy, this function has too wide a tail to ignore when calculating

$$\int_{-t}^{t} \frac{dx}{x^2 + 10^{-10}}$$

if t is large. In general, the compressing substitution $x = b \tan u$ maps the entire real line into $(-\pi/2, \pi/2)$ and maps subsets of the real line into subsets of $(-\pi/2, \pi/2)$. For $b = 1\text{E}{-5}$ the substitution becomes $x = 1\text{E}{-5} \tan u$ and the integral becomes

$$10^5 \int_{\tan^{-1}(-t/b)}^{\tan^{-1}(t/b)} du$$

which is readily computed for very large t.

This compressing substitution is also a standard way to deal with infinite intervals. For example,

$$\int_{-\infty}^{\infty} \frac{dx}{x^2 + 10^{-10}} = 10^5 \int_{-\pi/2}^{\pi/2} du$$

In some cases the tail can be chopped off. Consider the function $\exp(-x^2)$. This functions underflows (that is, gives a result of zero in machine arithmetic) for $x > 34$. Thus,

$$\int_0^{\infty} e^{-x^2} dx \approx \int_0^{34} e^{-x^2} dx$$

Therefore, when dealing with infinite integrals you can cut off the tail if it is insignificant, but you should use a compressing substitution if it is not.

About the Algorithm

The Math Pac uses a Romberg method for accumulating the value of an integral. Several refinements make it more effective. Instead of equally spaced samples, which can introduce a kind of resonance or aliasing that produces misleading results when the integrand is periodic, INTEGRAL uses samples that are spaced nonuniformly. Their spacing can be demonstrated by substituting

$$x = \frac{3}{2}u - \frac{1}{2}u^3 \text{ into } \int_a^b f(x)dx$$

and then spacing u uniformly. Besides suppressing resonance, the substitution has two additional benefits. First, no sample need be taken from either endpoint of the interval of integration unless the interval is so small that points in the interval round to an endpoint. As a result, an integral like

$$\int_0^1 \frac{\sin x}{x}\,dx$$

will not be interrupted by division by zero at an endpoint. Second, INTEGRAL can integrate functions whose slope is infinite at an endpoint. Such functions are encountered when calculating the area enclosed by a smooth closed curve like $x^2 + f^2(x) = R$.

In addition, INTEGRAL uses extended precision. Internally, sums are accumulated in 15-digit numbers. This allows thousands of samples to be accumulated, if necessary, without losing any more significance to round-off than is lost within your function.

During the computation, INTEGRAL generates a sequence of iterates that are increasingly accurate estimates of the actual value of the integral. It also estimates the width of the error ribbon at each iterate. INTEGRAL stops only after three successive iterates are within the computed error of each other or after 16 iterations have been performed without this criterion being met.

In the latter case the function will have been sampled at 65,535 points. The value returned by IBOUND will be the negative of the computed error to signify that the returned value of the INTEGRAL is likely not within the error tolerance of the actual value. Typically, you should then split up the interval of integration into smaller subintervals and integrate the function over each of the subintervals. The integral over the original interval will then be the sum of the integrals over the subintervals. In this way, up to 65,535 points can be sampled on each subinterval, thus computing the integral to greater precision.

In summary, INTEGRAL has been designed to return reliable results rapidly and in a convenient, easy-to-use fashion. The above theoretical considerations discuss problems with numerical integration in general. The INTEGRAL keyword is capable of handling even difficult integrals with relative ease.

Section 12

Finding Roots of Polynomials

Keyword

The keyword in this section finds *all* solutions—both real and complex—of $P(x) = 0$, where P is a polynomial of your choice with real coefficients. If P is a polynomial of degree n there will be n (not necessarily distinct) solutions of this equation, so this keyword resembles an array operation in its format.

To use this keyword to find the solutions of the equation $P(x) = 0$, where

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

first store the coefficients a_n, a_{n-1}, ..., a_0 in a real-type array with $n + 1$ elements in all. They should be stored in the order indicated above, with the coefficient of the highest power first and the constant term last. Aside from the total number of elements in the array, which indicates to the Math Pac the degree of the polynomial, the dimensions of the array are irrelevant. For example, the arrays

$$[6, 5, 4, 3, 2, 1], \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 6 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}$$

all can represent the fifth degree polynomial

$$6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1.$$

The array in which you wish the resulting roots to be stored must be complex type to accept complex roots. If the polynomial whose roots are sought has degree N, and if the result array is a vector, it will be redimensioned to have N elements. If the result array is a matrix, it will be redimensioned to have N rows and one column.

The degree of the polynomial whose roots you can find is limited only by the amount of memory you have available.

PROOT

Roots of ■ Polynomial

```
MAT R=PROOT(P)
```

where **P** is a real vector or matrix with $N + 1$ elements, where N = degree of polynomial whose roots are sought, and **R** is a complex vector or matrix.

If **R** is a vector, implicitly redimensions ■ to have N elements. If **R** is a matrix, implicitly redimensions ■ to have N rows and one column. ■ will be assigned the (complex) values of the solutions of the equation $P(x) = 0$ (where P is the polynomial of degree N whose coefficients are the values of the elements of **P**).

To halt operation, press ATTN twice.

Not usable in CALC mode.

Example

The following example finds all roots of the polynomial

$$5Z^6 - 45Z^5 + 225Z^4 - 425Z^3 + 170Z^2 + 370Z - 500$$

```
OPTION BASE 1 END LINE
```

```
DIM A(7) END LINE
```

Creates real vector for coefficients.

```
COMPLEX B(10) END LINE
```

Creates complex vector for roots.

```
MAT INPUT A END LINE
```

```
A(1)? ■
```

```
5,-45,225,-425,170,370,-500
END LINE
```

```
MAT B=PROOT(A) END LINE
```

First redimensions the vector **B** to have six elements, just large enough to contain the six (complex) roots of the six-degree polynomial. Then finds all roots and stores them in **B**.

```
MAT DISP B END LINE
```

Displays all roots.

```
 (1,1)
 (1,-1)
 (-1,0)
 (2,0)
 (3,4)
 (3,-4)
```

Additional Information

There are several methods of gauging the accuracy of the calculated roots. The first method is to calculate the value of the polynomial at the alleged root, and compare this value with zero. Although quite straightforward in theory, this has a number of drawbacks in practice. It may easily happen that the root calculated is the closest machine-representable number to a true root, but because the polynomial has such a large value for its derivative at this root, the value of the polynomial at the calculated root is very large. A simple example of this phenomenon is given by the polynomial $1E20x^2 - 2E20$. A true root is $\sqrt{2}$; a calculated root is 1.41421356237, which is the machine-representable number closest to $\sqrt{2}$. However, the value of the polynomial at this approximation to the square root of 2 is $-1{,}000{,}000{,}000$, a number that *seems* very far from zero.

Another drawback of the above method is that because of the limited precision available in any numerical calculation, the roundoff errors that occur in the calculation of the polynomial value may completely eliminate the significance of the difference between the calculated value and zero. This is especially true when the polynomial is of large degree, has coefficients widely varying in size, or has roots of high multiplicity.

A second method of gauging the accuracy of the calculated roots is to attempt to reconstruct the polynomial from these roots. If the reconstructed polynomial closely resembles the original, the roots are then judged to be accurate. This technique is less sensitive to the problems that affect the polynomial evaluation method. Of course, this method does not give information on the accuracy of an individual root.

The program below asks you for a polynomial and calculates the roots of that polynomial using the PROOT keyword. If you wish, the program will also calculate the reconstructed coefficients from the calculated roots. In addition, if desired, the program will compute the value of the polynomial at either a calculated root or any other real or complex value.

Lines 10 through 200 drive the program and use the PROOT function to calculate the roots of the given polynomial. Lines 210 through 250 comprise the subroutine that evaluates the polynomial at any real or complex point. Horner's method is used.

Lines 260 through 410 comprise the subroutine that reconstructs the coefficients from the calculated roots. Starting with the polynomial 1, the subroutine successively multiplies the polynomial by the linear factors $(Z - R)$, where R is a calculated real root, or by the quadratic $Z^2 - 2\text{REPT}(R) + \text{ABS}(R)^2$, where R is a calculated complex root. (Note that CONJ(R) will also be a calculated root).

```
10 OPTION BASE 0 @ INTEGER D,E
   @ DIM U$[4] @ DELAY 1 @ WIDTH 96
20 INPUT "DEGREE? ";D
30 DIM P(D),C(D) @ COMPLEX R(D-1)
40 DISP "ENTER COEFFICIENTS "
   @ MAT INPUT P
50 DISP "WORKING..."
60 MAT R=PROOT(P)
70 DISP "THE ROOTS ARE" @ DELAY 8 @
   MAT DISP R @ DELAY 1
80 U$=KEY$ @ INPUT
   "RECONSTRUCT? (Y/N) ";U$
90 IF UPRC$(U$)="Y" THEN GOSUB 260
   ELSE 110
100 DISP "RCON COEFFICIENTS ARE" @
    DELAY 8 @ MAT DISP C @ DELAY 1
110 U$=KEY$ @ INPUT
    "EVALUATION? (Y/N) ";U$
120 IF UPRC$(U$)#"Y" THEN 190
    ELSE COMPLEX Z
130 INPUT "AT A ROOT? (Y/N) ";U$
140 IF UPRC$(U$)#"Y" THEN INPUT
    "VALUE? ";Z @ GOTO 160
```

D is the degree of the polynomial.

Array **P** will contain the coefficients of the polynomial in the order given previously, array **R** will contain the calculated roots, and array **C** will contain the reconstructed coefficients.

Enter the coefficients. The leading coefficient should be nonzero for the program to work properly.

Calculates the roots and stores them in array **R**.

Displays the calculated roots. To continue the program after each root is displayed, press END LINE.

If you wish, the program will reconstruct the coefficients from the calculated roots.

The subroutine starting at line 260 performs the reconstruction and stores the reconstructed coefficients in array **C**.

Displays the reconstructed coefficients. To continue the program after each display, press END LINE.

If you wish, the program will evaluate the polynomial at a root or at any other point.

The complex variable Z will hold the polynomial value.

The point may be either real or complex.

```
150 DISP USING '#,"WHICH ROOT
    (1...",K,")";D @ INPUT E
    @ Z=R(E-1)
```

Input the number of the root where you want the polynomial evaluated.

```
160 GOSUB 210 @ DISP "POLYNOMIAL
    VALUE IS" @ DELAY 8 @ DISP Z @
    DELAY 1
```

The subroutine beginning at line 210 evaluates the polynomial at the given point or root. This value is then displayed. To continue, press END LINE.

```
170 U$=KEY$ @ INPUT
    "ANOTHER VALUE? (Y/N) ";U$
```

The program will evaluate the polynomial again if you wish.

```
180 IF UPRC$(U$)="Y" THEN 130
190 INPUT "ANOTHER POLY? (Y/N) ";U$
```

You can choose to start the program over again with a new polynomial.

```
200 IF UPRC$(U$)="Y" THEN 20 ELSE STOP
210 COMPLEX B @ B=P(0)
```

The polynomial evaluation subroutine uses Horner's method.

```
220 FOR K=1 TO D
230 B=P(K)+Z*B
240 NEXT K
250 Z=B @ DESTROY B @ RETURN
260 DISP "WORKING..."
```

This line begins the coefficient reconstruction subroutine. Some rounding error may accumulate during reconstruction, so even if the roots are exact, the reconstructed coefficients may not exactly coincide with the original coefficients.

```
270 MAT C=ZER @ C(D)=1
```

Creates polynomial 1 in array C.

```
280 FOR L=1 TO D
```

We use each calculated root in turn.

```
290 IF IMPT(R(L-1))#0 THEN 340
300 FOR K=D-L TO D-1
310 C(K)=C(K+1)-C(K)*REPT(R(L-1))
320 NEXT K
330 C(D)=-C(D)*REPT(R(L-1)) @ GOTO
    400
```

Lines 300 through 330 multiply the current reconstructed polynomial by a linear factor.

```
340 REAL B @ B=REPT(R(L-1))^2
    +IMPT(R(L-1))^2
350 FOR K=D-L-1 TO D-2
```

Lines 340 through 390 multiply the current reconstructed polynomial by a quadratic factor.

```
360 C(K)=C(K+2)-2*REPT(R(L-1))
    *C(K+1)+B*C(K)
370 NEXT K
380 C(D-1)= -2*REPT(R(L-1))*C(K+1)
    +B*C(K)
390 C(D)=B*C(D) @ L=L+1
400 NEXT L
410 MAT C=(P(0))*C @ DESTROY B
    @ RETURN
```

We increment L since we multiplied the polynomial by both the complex root and its complex conjugate.

The reconstructed polynomial has leading coefficient 1 and so must be adjusted if the original leading coefficient was not 1.

Example. If we wanted to find and evaluate the roots of the polynomial

$$x^6 + x^5 + x^4 + x^3 + x^2 + x + 1,$$

we would run the program using the following keystrokes.

Input/Result

RUN

```
DEGREE? ■
```

6 END LINE

```
ENTER COEFFICIENTS
```

```
P(0)? ■
```

1 , 1 , 1 , 1 , 1 , 1 , 1 END LINE

```
WORKING...
```

```
 THE ROOTS ARE
```

```
 33956,-.974927912182)
```

The display scrolls to display the imaginary part of the first root.

g ◄

```
 (-.222520933956,-.974
```

The real part of the first root.

END LINE

```
 933956,.974927912182)
```

The imaginary part of the second root.

g ◄

```
 (-.222520933956,.9749
```

The real part of the second root.

Display the last four roots in the same way. These displayed roots are:

Third root: (-.900968867902,-.433883739118)

Fourth root: (-.900968867902,.433883739118)

Fifth root: (.623489801859,.781831482468)

Sixth root: (.623489801859,-.781831482468)

After the last root is displayed, continue the program by pressing END LINE.

Input/Result

```
 RECONSTRUCT? (Y/N) ■
```

Any response but Y or y is interpreted as "no."

Y END LINE

```
 WORKING...
```

```
RCON COEFFICIENTS ARE
```

```
 1
```

The coefficient of the x^6 term.

END LINE

```
 .99999999999
```

The coefficient of the x^5 term.

Display the remaining five coefficients in the same way. These displayed coefficients are:

Coefficient of x^4 term: 1

Coefficient of x^3 term: .99999999998

Coefficient of x^2 term: 1

Coefficient of x^1 term: .99999999999

Coefficient of x^0 term: 1

After the last coefficient is displayed, continue the program by pressing END LINE.

Input/Result

```
EVALUATION? (Y/N) ■
```

Y END LINE

```
AT A ROOT? (Y/N) ■
```

Y END LINE

```
WHICH ROOT (1...6)? ■
```

1 END LINE

```
POLYNOMIAL VALUE IS

(0,-7.52E-13)
```

END LINE — Continues the program.

```
ANOTHER VALUE? (Y/N) ■
```

Y END LINE

```
AT A ROOT? (Y/N) ■
```

N END LINE

```
VALUE? ■
```

(-.2,.9) END LINE

```
POLYNOMIAL VALUE IS

(.222523,.185814)
```

END LINE

```
ANOTHER VALUE? (Y/N) ■
```

N END LINE

```
ANOTHER POLY? (Y/N) ■
```

N END LINE — Ends the program.

About the Algorithm

The Math Pac finds the roots of polynomials using Laguerre's method, which is an iterative process. The Laguerre step at the iterate Z_k for the polynomial $P(Z)$ of degree N is

$$\frac{-NP(Z_k)}{P'(Z_k) \pm [(N-1)^2 (P'(Z_k))^2 - N(N-1) P(Z_k) P''(Z_k)]^{1/2}}$$

The sign in the denominator is chosen to give the Laguerre step of smaller magnitude. Polynomials or their quotients of degree < 3 are solved using the quadratic formula or linear factorization.

Laguerre's method is cubically convergent to isolated zeros and linearly convergent to zeros of multiplicity greater than one.

The PROOT function is global in the sense that the user is not required to supply either an initial guess or a stopping criterion; in other words, no prior knowledge of the location of the roots is assumed. The PROOT function always attempts to begin its search (iteration) at the origin of the complex plane. An annulus in the plane known to contain the smallest magnitude root of the current (original or quotient) polynomial is constructed about the origin (using five theoretical bounds) and the initial Laguerre step is rejected if it exceeds the upper limit of this annulus. In this case, a spiral search from the lower radius of the annulus in the direction of the rejected initial step is begun until a suitable initial iterate is found.

Once the iteration process has successfully started, circles around each iterate are constructed (using two theoretical bounds) that are known to bound the root closest to that iterate; the Laguerre step size is constantly tested against the radii of these circles and modification of the step is made when it is deemed to be too large or when the polynomial value does not decrease in the direction of the step. For this reason, the roots are normally found in order of increasing magnitude, thus minimizing the roundoff errors resulting from deflation.

Evaluation of the polynomial and its derivatives at a real iterate is exactly Horner's method. Evaluation at a complex iterate is a modification of Horner's method that saves approximately half of the multiplications. This modification takes advantage of the fact that the Horner recurrence is symmetric with respect to complex conjugation.

PROOT uses a sophisticated technique to determine when an approximation Z_k should be accepted as a root. As the polynomial is being evaluated at Z_k, a bound for the evaluation roundoff error is also being computed. If the polynomial value is less than this bound, Z_k is accepted as a root. Z_k can also be accepted as a root if the value of the polynomial is decreasing but the size of the Laguerre step has become negligible. Before an approximation Z_k is used in an evaluation, its imaginary part is set to zero if this part is small compared to the step size. This improves performance, since real-number evaluations are faster than complex evaluations. If the Laguerre step size has become negligible but the polynomial is not decreasing, then the message `PROOT failure` is reported and the computation stops. This is expected never to occur in practice.

As the polynomial is being evaluated, the coefficients of the quotient polynomial (by either a linear or quadratic factor corresponding to the Z_k) are also computed. When an approximation Z_k is accepted as a root, this quotient polynomial becomes the polynomial whose roots are sought, and the process begins again.

Multiple Zeros

No polynomial rootfinder, including PROOT, can consistently locate zeros of high multiplicity with arbitrary accuracy. The general rule-of-thumb for PROOT is that for multiple or nearly-multiple zeros, resolution of the root is approximately $12/K$ significant digits, where K is the multiplicity of the root.

Accuracy

PROOT's criterion for accuracy is that the coefficients of the polynomial reconstructed from the calculated roots should closely resemble the original coefficients.

We will illustrate PROOT's performance with isolated zeros using the 100th degree polynomial

$$P(Z) = \sum_{k=0}^{100} Z^k$$

Of the 200 real and imaginary components of the calculated roots, about half were found to 12 digit accuracy. Of the rest, the error did not exceed a few counts in the 12th digit.

The polynomial $(Z + 1)^{20}$ with all 20 roots equal to -1 was solved by PROOT to yield the following roots.

(−.997874038627,0)
(−.934656570635,0)
(−.947080146258,−.160105886062)
(−.947080146258,.160105886062)
(−.678701343788,−6.24034855342E−2)
(−.678701343788,6.24034855342E−2)
(−.815082852233,−.270565874916)
(−.815082852233,.270565874916)
(−.725960092383,−.178602450179)
(−.725960092383,.178602450179)
(−.934932478844,−.326980158732)
(−.934932478844,.326980158732)
(−1.06905713438,−.337946194292)
(−1.06905713438,.337946194292)
(−1.19977533452,−.295162714497)

(−1.19977533452,.295162714497)
(−1.30383056467,−.200016185042)
(−1.30383056467,.200016185042)
(−1.3593147483,7.00833934259E−2)
(−1.3593147483,−7.00833934259E−2)

The roots appear inherently inaccurate due to the high multiplicity of −1 as a root. Between 0 and 1 correct digits were expected, even though the first zero found was better than this. However, the reconstructed coefficients are very close and are shown below (rounded to 12 digits).

Original Coefficients	Reconstructed Coefficients
1	1
20	20
190	190.000000001
1140	1140
4845	4845.00000003
15504	15504
38760	38760.0000003
77520	77520.0000007
125970	125970.000001
167960	167960.000002
184756	184756.000002
167960	167960.000003
125970	125970.000002
77520	77520.0000015
38760	38760.0000009
15504	15504.0000004
4845	4845.00000011
1140	1140.00000004
190	190.000000042
20	20.0000000344
1	1.00000001018

Time Performance

The speed of the PROOT function is illustrated in the following table. The times given are those required to calculate *all* the roots of the polynomial

$$P(Z) = \sum_{k=0}^{N} Z^k$$

for values of N given in the Degree column.

Note that times are approximate.

Degree	Time (sec)
3	3
5	6
10	22
15	42
20	142
30	168
50	568
70	1060
100	2101

Section 13

Finite Fourier Transform

Keyword

The finite Fourier transform is a key step in solving many problems in mathematics, physics, and engineering, such as problems in signal processing and differential equations.

Given a set of N complex data points $Z_0, Z_1, \ldots, Z_{N-1}$, the finite Fourier transform will return another set of N complex values $W_0, W_1, \ldots, W_{N-1}$, such that for $k = 0, 1, \ldots, N-1$,

$$Z_k = \sum_{j=0}^{N-1} W_j \left(\cos \frac{2\pi kj}{N} + i \sin \frac{2\pi kj}{N} \right)$$

The W's then represent the complex amplitudes of the various frequency components of the signal represented by the data points. The values for the W's are given by the formula

$$W_j = 1/N \sum_{k=0}^{N-1} Z_k \left(\cos \frac{-2\pi kj}{N} + i \sin \frac{-2\pi kj}{N} \right)$$

This formula holds for any number of data points. The Math Pac uses the Cooley-Tukey algorithm and the internal language of the HP-71 to achieve excellent speed and accuracy in the calculation of the finite Fourier transform. This requires, however, that N be an integral power of 2; for example, 2, 4, 8, 16, 32, 64, and 128 are all acceptable values for the number of complex data points.

To use the finite Fourier transform, store your complex data points $Z_0, \ldots, Z_{N-1}$ as successive elements of an N-element complex array with Z_0 as the first element, Z_1 as the second element, and so on. Aside from the total number of elements in the array, which indicates to the Math Pac the number of complex data points, the dimensions of the array are irrelevant. For example, each of the following eight-element arrays

$$\begin{bmatrix} (1,2) \\ (3,4) \\ (5,6) \\ (7,8) \\ (9,10) \\ (11,12) \\ (13,14) \\ (15,16) \end{bmatrix}$$

$$\begin{bmatrix} (1,2) & (3,4) \\ (5,6) & (7,8) \\ (9,10) & (11,12) \\ (13,14) & (15,16) \end{bmatrix}$$

$$\begin{bmatrix} (1,2) & (3,4) & (5,6) & (7,8) \\ (9,10) & (11,12) & (13,14) & (15,16) \end{bmatrix}$$

$$\begin{bmatrix} (1,2) & (3,4) & (5,6) & (7,8) & (9,10) & (11,12) & (13,14) & (15,16) \end{bmatrix}$$

can represent the set of input data points

$$\{(1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(13,14),(15,16)\}$$

The array in which you wish the transformed data to be stored must be complex type. If the number of input data points is N, and if the result array is a vector, it will be redimensioned to have N elements. If the result array is a matrix, it will be redimensioned to have N rows and one column. The results of the finite Fourier transform $W_0,\ldots,W_{N-1}$ will be returned with the complex values stored in successive elements of this N-element complex result array—the same form as the data points.

The number of data points you can use is limited only by the amount of available memory and by the requirement that the number of data points be a non-negative integral power of 2.

FOUR

Finite Fourier Transform

```
MAT W=FOUR(Z)
```

where **Z** is an *N*-element complex array, either a vector or matrix, *N* is the number of complex data points, which must be a non-negative integer power of 2, and **W** is a complex array, either a vector or matrix.

If **W** is a vector, implicitly redimensions **W** to have *N* elements; if **W** is a matrix, implicitly redimensions **W** to have *N* rows and one column. **W** will be assigned the complex values of the finite Fourier transform of the data points represented by **Z**.

To halt operation, press ATTN twice.

Not usable in CALC mode.

Example

The following example computes the finite Fourier transform of the input data set ((1,2), (3,4), (5,6), (7,8), (9,10), (11,12), (13,14), (15,16)).

```
10 OPTION BASE 1
20 COMPLEX SHORT A(8),B(1,2)

30 MAT INPUT A
40 MAT B=FOUR(A)
50 MAT DISP B
```

A contains the data set, and **B**, after redimensioning, contains the transform of the data.

RUN

```
A(1)? ■
```

```
(1,2),(3,4),(5,6),(7,8),(9,10),
(11,12),(13,14),(15,16)
END LINE
```

```
(8,9)
(-3.4142,1.4142)
(-2,0)
(-1.4142,-.58579)
(-1,-1)
(-.58579,-1.4142)
(0,-2)
(1.4142,-3.4142)
```

Additional Information

Time Performance

The approximate time required by FOUR to return the transform, based on the number of data points, is shown in this table.

Number of Data Points	Transform Time (Seconds)
1	0.07
2	0.11
4	0.26
8	0.75
16	1.9
32	4.7
64	11
128	25
256	55
512	120
1024	260
2048	558

Relation Between the Finite and Continuous Fourier Transform

The finite Fourier transform is most often used as an approximation to the continuous (infinite) Fourier transform. To understand in what sense it is an approximation, and to understand the effects of various choices to be made in using this approximation, it is most useful to have the direct relationship between the continuous and finite transforms.

If $Z(x)$ is a complex valued function, its continuous Fourier transform is defined to be

$$W(f) = \int_{-\infty}^{\infty} Z(x) \exp(-2\pi i f x)\, dx$$

If we have a set of N complex data points $Z_0, Z_1, \ldots, Z_{N-1}$ given by sampling the function Z at N equally spaced points

$$Z_k = Z(x_0 + k\Delta x) \text{ for } k = 0, 1, \ldots, N-1,$$

and then find the finite Fourier transform $W_0, W_1, \ldots, W_{N-1}$ of this data set, we can relate these values to the values of the continuous Fourier transform $W(f)$ as follows. For $k = 0, \ldots, N-1$,

$$W_k = (r/N)\ \tilde{W}(k/N\Delta x) \text{ where } r = \exp(-2\pi i x_0).$$

$\tilde{W}$ is a "smeared" version of the true continuous Fourier transform W. To get $\tilde{W}$ from W, you must average W in two important but very different ways. The first type of averaging that occurs can be described by defining a new function $A(f)$ intermediate between W and $\tilde{W}$.

$$A(f) = \sum_{k=-\infty}^{\infty} W(f + k/\Delta x)$$

This says that the value of A at a point f is equal to the sum of the values of W at all points that are integer multiples of the limiting frequency $1/\Delta x$ away from f. In particular, if W consists of a small bump centered at the origin, then A will consist of an infinite sequence of bumps spaced $1/\Delta x$ units apart. This is the aspect of the finite Fourier transform that gives rise to *aliasing*: any frequency that occurs in W (that is, W has a nonzero value there) will give rise to a nonzero value for A (and also $\tilde{W}$) somewhere in the interval $[0, 1/\Delta x]$ *no matter what the original frequency was*. For this reason, you should choose Δx small enough so that $1/\Delta x$ is larger than the distance between the largest and smallest f's that you suspect will occur in W. Since most functions occuring in actual situations (and *all* real-valued functions) have continuous Fourier transforms that are roughly symmetric about the origin, if a frequency f_0 occurs in W, it is likely that $-f_0$ also occurs in W. For the finite Fourier transform to contain both frequencies without aliasing, $1/\Delta x$ must be larger than $2f_0$. If we define the largest frequency occuring in W as Δf, we can express the no-aliasing requirement as $\Delta f \Delta x < 1/2$.

The second type of averaging that occurs when going between W and $\tilde{W}$ is much more local in nature than the first. It results in a loss of frequency resolution in $\tilde{W}$ as compared with W; more precisely,

$$\tilde{W}(f) = (N\Delta x) \int_{-\infty}^{\infty} \text{sinc}(gN\Delta x)\, A(f - g)\, dg$$

where $\text{sinc}(a) = \begin{cases} 1 \text{ if } a = 0, \\ \dfrac{\sin(\pi a)}{\pi a} \text{ otherwise.} \end{cases}$

Since sinc $(gN\Delta x)$ consists primarily of a bump with width inversely proportional to $N\Delta x$, $\tilde{W}$ is more blurred (compared to W) for smaller values of $N\Delta x$. This is not a serious problem unless W has a large value at a frequency that is not a multiple of the fundamental frequency $N/\Delta x$. In this case, the "side lobes" of the sinc function become evident in $\tilde{W}$. This can be reduced somewhat by multiplying the data values Z_k by a smoothing function $G(k)$ before taking the finite Fourier transform. This results in an averaging function that has smaller side lobes than the sinc function. One example of such a function is the Hanning function $G(k) = (1/2)(1 - \cos(2\pi k/N))$.

Inverse Finite Fourier Transform

Many applications of the finite Fourier transform involve taking the transform of a set of data points, operating on the transformed values (for example, increasing or decreasing the amplitudes), and then retransforming the data using the inverse Fourier transform defined by

$$Z_k = \sum_{j=0}^{N-1} W_j \left(\cos \frac{2\pi kj}{N} + i \sin \frac{2\pi kj}{N}\right)$$

You can also use the FOUR keyword to compute the inverse finite Fourier transform in a simple way. If **W** is an N-element complex array for which you want the inverse finite Fourier transform:

1. Redimension **W** to have N rows and one column (if **W** is an array with only one column, then no redimensioning is necessary).
2. Take the transpose (TRN) of **W**. This produces the complex conjugate of **W**, without changing the order of the elements.
3. Take the finite Fourier transform of the result.
4. Take the transpose of the result of the finite Fourier transform and scalar multiply this result by N. This will produce the inverse finite Fourier transform of the original array.

Example. This illustrates an application of the finite Fourier transform, and shows the procedure for obtaining the inverse finite Fourier transform.

Suppose we want to find the steady state solution $Z(x)$ of the inhomogeneous differential equation

$$Z''(x) + 3Z'(x) + 12Z(x) = P(x)$$

where $P(x)$ is a function for which we have sampling data. If we denote the (continuous) Fourier transform of any function Q by $\tilde{Q}$, by taking the Fourier transform of the above equation we arrive at

$$-f^2\tilde{Z}(f) + 3if\tilde{Z}(f) + 12\tilde{Z}(f) = \tilde{P}(f).$$

Solving this equation algebraically we obtain

$$\tilde{Z}(f) = \frac{\tilde{P}(f)}{(-f^2 + 12) + 3if}$$

If we can get a good approximation of $\tilde{P}$, we can easily calculate the right hand side of this equation. From this result we can obtain the solution to the original equation by taking the inverse Fourier transform.

For simplicity, we will assume that the equation has been scaled so the $P(x)$ has unit period, and that the highest frequency component of P is (approximately) 30 times the fundamental frequency. Sampling P 64 times in one period will then suffice to avoid aliasing.

Rather than prompt the user for 64 complex data points representing the sampling of P, the program below uses a relatively simple function for P, although you could use values from any other source equally well.

```
10 OPTION BASE 1
20 COMPLEX P(64),Q(64,1),Z(1,64)
30 COMPLEX T
40 DISP "Working; please wait."
50 RADIANS
```

P will contain the data points representing the sampling of P. **Q** will represent $\tilde{P}$ and eventually $\tilde{P}/(-f^2 + 3if + 12)$. **Z** will represent the solution to the differential equation.

T is a complex scalar for use in the complex division.

```
60 FOR I=1 TO 64
70 R=PI*I/32
80 P(I)=( 6000*COS(3*R)*SIN(7.5*R)*
     COS(5.5*R) , 4000*COS(13*R)+
     3500*SIN(11*R) )
90 NEXT I
100 MAT Q=FOUR(P)
110 FOR F=-31 TO 32
120 J=MOD(F,64)+1
130 T=(-F^2+12,3*F)
140 Q(J,1)=Q(J,1)/T
150 NEXT F
160 MAT Q=TRN(Q)
170 MAT Z=FOUR(Q)
180 MAT Z=TRN(Z)
190 MAT Z=(64)*Z
200 COMPLEX Z(64,1)
210 DISP "The values are"
220 MAT DISP USING
     "X,C(MDDD.D,MDDD.D)";Z
```

This is the sampling routine that assigns to **P** the values of the complex-valued functions represented by the right-hand side of line 80, sampled at 64 equally spaced points.

Q now represents $\tilde{P}$.

F represents the frequency variable and spans the full range of frequencies, positive and negative, that we expect to occur in $\tilde{P}$.

J represents the number of the element in the **Q** array where the amplitude of the frequency F is stored.

T will be the denominator of the complex fraction.

■ now represents $\tilde{P}/(-f^2 + 3if + 12)$.

This starts the procedure that assigns the values of the inverse Fourier transform to **Z**. The transpose is used here to take the conjugate of **Q**.

The transpose is used here for conjugation as well.

The values displayed will represent the complex values of the steady state solution of the differential equation sampled at 64 equally spaced points in one period.

Fourier Sine/Cosine Series

There is another transform closely related to the finite Fourier transform that is applicable when the data points Z_k are purely real (that is, their imaginary parts are equal to zero). This is the Fourier series transformation, which takes a set of $2N$ (real) data points $Z_0, Z_1, \ldots, Z_{2N-1}$ and returns a set of $2N + 1$ real values $A_0, A_1, \ldots, A_N, B_1, \ldots, B_N$ with the property that

$$Z_k = \frac{A_0}{2} + \sum_{j=1}^{N} A_j \cos \frac{2\pi jk}{2N} + B_j \sin \frac{2\pi jk}{2N} .$$

If $W_0, W_1, \ldots, W_{2N-1}$ are the complex values of the fir te Fourier transform of the real data points $Z_0, \ldots, Z_{2N-1}$, then the Fourier series values are given by

$$A_j = 2\text{REPT}(W_j) \qquad \text{for } j = 0, \ldots, N-1,$$

$$A_N = \text{REPT}(W_N)$$

$$B_j = -2\text{IMPT}(W_j) \qquad \text{for } j = 1, \ldots, N.$$

Appendix A

Owner's Information

Installing and Removing the Math Pac Module

The math module can be plugged into any one of the four ROM ports on the front edge of the computer.

CAUTIONS

- Be sure to turn off the HP-71 (press f OFF) before installing or removing the module.
- If you have removed a module to make a port available for the math module, before installing the math module, turn the computer on and then off to reset internal pointers.
- Do not place fingers, tools, or other objects into any of the ports. Such actions could result in minor electircal shock hazard and interference with pacemaker devices worn by some persons. Damage to port contacts and internal circuitry could also result.
- If ■ module jams when inserted into a port, it may be upside down. Attempting to force it further may result in damage to the computer or the module.
- Handle the plug-in modules very carefully while they are out of the computer. Do not insert any objects in the module connecter socket. Always keep a blank module in the computer port when a module is not installed. Failure to observe these cautions may result in damage to the module or the computer.

Limited One-Year Warranty

What We Will Do

The Math Pac is warranted by Hewlett-Packard against defects in materials and workmanship affecting electronic and mechanical performance, but not software content, for one year from the date of original purchase. If you sell your unit or give it as ■ gift, the warranty is transferred to the new owner and remains in effect for the original one-year period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to a Hewlett-Packard service center.

What Is Not Covered

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than an authorized Hewlett-Packard service center.

No other express warranty is given. The repair or replacement of a product is your exclusive remedy.

ANY OTHER IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE ONE-YEAR DURATION OF THIS WRITTEN WARRANTY. Some states, provinces, or countries don't allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. **IN NO EVENT SHALL HEWLETT-PACKARD COMPANY BE LIABLE FOR CONSEQUENTIAL DAMAGES.** Some states, provinces, or countries do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which may vary from state to state, province to province, or country to country.

Warranty for Consumer Transactions in the United Kingdom

This warranty shall not apply to consumer transactions and shall not affect the statutory rights of a consumer. In relation to such transactions, the rights and obligations of Seller and Buyer shall be determined by statute.

Obligation To Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. Hewlett-Packard shall have no obligation to modify or update products once sold.

Warranty Information

If you have any questions concerning this warranty, please contact an authorized Hewlett-Packard dealer or a Hewlett-Packard sales and service office. Should you be unable to contact them, please contact:

- In the United States:

Hewlett-Packard Company
Personal Computer Group
Customer Communications
11000 Wolfe Road
Cupertino, CA 95014

Toll-Free Number: (800) FOR-HPPC (800 367-4772)

- In Europe:

Hewlett-Packard S.A.
150, route du Nant-d'Avril
P.O. Box CH-1217 Meyrin 2
Geneva
Switzerland
Telephone: (022) 83 81 11

Note: Do not send products to this address for repair.

- In other countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Rd.
Palo Alto, CA 94304
U.S.A.
Telephone: (415) 857-1501

Note: Do not send products to this address for repair.

Service

Service Centers

Hewlett-Packard maintains service centers in most major countries throughout the world. You may have your product repaired at a Hewlett-Packard service center any time it needs service, whether the unit is under warranty or not. There is ■ charge for repairs after the one-year warranty period.

Hewlett-Packard computer products normally are repaired and reshipped within five (5) working days of receipt at any service center. This is an average time and could vary depending on the time of year and work load at the service center. The total time you are without you product will depend largely on the shipping time.

Obtaining Repair Service in the United States

The Hewlett-Packard United States Service Center for battery-powered computational devices is located in Corvallis, Oregon:

Hewlett-Packard Company
Service Department
P.O. Box 999
Corvallis, OR 97339, U.S.A.
or
1030 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.
Telephone: (503) 757-2000

Obtaining Repair Service in Europe

Service centers are maintained at the following locations. For countries not listed, contact the dealer where you purchased your unit.

AUSTRIA
HEWLETT-PACKARD Ges.m.b.H.
Kleinrechner-Service
Wagramerstrasse-Lieblgasse 1
A-1220 Wien (Vienna)
Telephone: (0222) 23 65 11

BELGIUM
HEWLETT-PACKARD BELGIUM SA/NV
Woluwedal 100
B-1200 Brussels
Telephone: (02) 762 32 00

DENMARK
HEWLETT-PACKARD A/S
Datavej 52
DK-3460 Birkerod (Copenhagen)
Telephone: (02) 81 66 40

EASTERN EUROPE
Refer to the address listed under Austria.

FINLAND
HEWLETT-PACKARD OY
Revontulentie 7
SF-02100 Espoo 10 (Helsinki)
Telephone: (90) 455 02 11

FRANCE
HEWLETT-PACKARD FRANCE
Division Informatique Personnelle
S.A.V. Calculateurs de Poche
F-91947 Les Ulis Cedex
Telephone: (6) 907 78 25

GERMANY
HEWLETT-PACKARD GmbH
Kleinrechner-Service
Vertriebszentrale
Berner Strasse 117
Postfach 560 140
D-6000 Frankfurt 56
Telephone: (611) 50041

ITALY
HEWLETT-PACKARD ITALIANA S.P.A.
Casella postale 3645 (Milano)
Via G. Di Vittorio. 9
I-20063 Cernusco Sul Naviglio (Milan)
Telephone: (2) 90 36 91

NETHERLANDS
HEWLETT-PACKARD NEDERLAND B.V.
Van Heuven Goedhartlaan 121
N-1181 KK Amstelveen (Amsterdam)
P.O. Box 667
Telephone: (020) 472021

NORWAY
HEWLETT-PACKARD NORGE A/S
P.O. Box 34
Oesterndalen 18
N-1345 Oesteraas (Oslo)
Telephone: (2) 17 11 80

SPAIN
HEWLETT-PACKARD ESPANOLA S.A.
Calle Jerez 3
E-Madrid 16
Telephone: (1) 458 2600

SWEDEN
HEWLETT-PACKARD SVERIGE AB
Skalholtsgatan 9, Kista
Box 19
S-163 93 Spanga (Stockholm)
Telephone: (08) 750 20 00

SWITZERLAND
HEWLETT-PACKARD (SCHWEIZ) AG
Kleinrechner-Service
Allmend 2
CH-8967 Widen
Telephone: (057) 31 21 11

UNITED KINGDOM
HEWLETT-PACKARD Ltd
King Street Lane
GB-Winnersh, Wokingham
Berkshire RG11 5AR
Telephone: (0734) 784 774

International Service Information

Not all Hewlett-Packard service centers offer service for all models of HP products. However, if you bought your product from an authorized Hewlett-Packard dealer, you can be sure that service is available in the country where you bought it.

If you happen to be outside of the country where you bought your unit, you can contact the local Hewlett-Packard service center to see if service is available for it. If service is unavailable, please ship the unit to the address listed above under Obtaining Repair Service in the United States. A list of service centers for other countries can be obtained by writing to that address.

All shipping, reimportation arrangements, and customs costs are your responsibility.

Service Repair Charge

There is a standard repair charge for out-of-warranty repairs. The repair charges include all labor and materials. In the United States, the full charge is subject to the customer's local sales tax.

Computer products damaged by accident or misuse are not covered by the fixed repair charge. In these cases, repair charges will be individually determined based on time and materials.

Service Warranty

Any out-of-warranty repairs are warranted against defects in materials and workmanship for a period of 90 days from date of service.

Shipping Instructions

Should your product require service, return it with the following items:

- A completed Service Card, including a description of the problem.
- A sales receipt or other documentary proof of purchase date if the one-year warranty has not expired.

The product, the Service Card, a brief description of the problem, and (if required) the proof of purchase date should be packaged in adequate protective packaging to prevent in-transit damage. Such damage is not covered by the one-year limited warranty; Hewlett-Packard suggests that you insure the shipment to the service center. The packaged product should be shipped to the nearest Hewlett-Packard designated collection point or service center. Contact your dealer for assistance.

Whether the product is under warranty or not, it is your responsibility to pay shipping charges for delivery to the Hewlett-Packard service center.

After warranty repairs are completed, the service center returns the product with postage prepaid. On out-of-warranty repairs in the United States and some other countries, the product is returned C.O.D. (covering shipping costs and the service charge).

Further Information

Service contracts are not available. Computer products circuitry and design are proprietary to Hewlett-Packard, and service manuals are not available to customers. Should other problems or questions arise regarding repairs, please call your nearest Hewlett-Packard service center

When You Need Help

Hewlett-Packard is committed to providing after-sale support to its customers. To this end, our customer support department has established phone numbers that you can call if you have questions about this product.

Product Information. For information about Hewlett-Packard dealers, products, and prices, call the toll-free number below:

(800) FOR-HPPC
(800 367-4772)

Technical Assistance. For technical assistance with your product, call the number below:

(503) 754-6666

For either product information or technical assistance, you can also write to:

Hewlett-Packard Company
Personal Computer Group
Customer Communications
11000 Wolfe Road
Cupertino, CA 95014

Appendix B

Memory Requirements

The Math Pac reserves 43.5 bytes of read/write memory for its own uses. In addition, small amounts of memory are temporarily used for routine overhead purposes. Significant amounts of memory can be used to declare complex variables and arrays (see page 20), and to redimension arrays to a larger size, but this memory usage is easily determined. This appendix lists the amounts of tempory memory used by other Math Pac operations.

Item	Memory Required For Operation
Matrix operations	
DET(**A**)	$2N(4N + 1)$ bytes, where **A** is an $N \times N$ matrix.
MAT PRINT USING	14 bytes.
MAT DISP USING	14 bytes.
MAT INPUT	40 bytes.
MAT **A**=**A*****A** MAT **A**=**A*****B** MAT **A**=**B*****A**	Requires additional memory only if an operand array is used for the result array. If the product (that is, the redimensioned array **A**) is $M \times N$ (for vectors, let $N = 1$), then the memory required is: 3*MN* bytes, if **A** is type INTEGER. 4.5*MN* bytes, if **A** is type SHORT. 8*MN* bytes, if **A** is type REAL. 9*MN* bytes, if **A** is type COMPLEX SHORT, 16*MN* bytes, if **A** is type COMPLEX.
MAT **A**=TRN(**A**)***A** MAT **A**=TRN(**A**)***B** MAT **A**=TRN(**B**)***A**	Requires additional memory only if an operand array is used for the result array. If the product (that is, the redimensioned array **A**) is $M \times N$ (for vectors, let $N = 1$), then the memory required is: 3*MN* bytes, if **A** is type INTEGER. 4.5*MN* bytes, if **A** is type SHORT. 8*MN* bytes, if **A** is type REAL. 9*MN* bytes, if **A** is type COMPLEX SHORT. 16*MN* bytes, if **A** is type COMPLEX.

Item	Memory Required For Operation
MAT **B**=INV(**A**)	**A** is $N \times N$. If **A** is REAL, SHORT or INTEGER and **B** is REAL: $4N$ bytes. If **A** is REAL, SHORT or INTEGER and **B** is *not* REAL: $4N(2N + 1)$ bytes. If **A** is COMPLEX or COMPLEX SHORT: $8N(4N + 1)$ bytes.
MAT **C**=SYS(**A**,**B**)	**A** is $N \times N$ and **B** is $N \times P$ (for vectors, let $P = 1$). If **A** is REAL, SHORT, or INTEGER and **B** is REAL, SHORT, or INTEGER: $4N(2N + 4P + 1)$ bytes. If **A** is REAL, SHORT, or INTEGER and **B** is COMPLEX or COMPLEX SHORT: $4N(2N + 8P + 1)$ bytes. If **A** is COMPLEX or COMPLEX SHORT: $8N(4N + 4P + 1)$ bytes.
MAT **A**=TRN(**A**)	If **A** is $M \times N$ and INTEGER: $MN/2$ bytes. If operand and result matrix are different, or if **A** is not INTEGER, no extra memory is needed.
MAT **B**=PROOT(**A**)	**A** represents an Nth degree polynomial. $21N + 261$ bytes.
MAT **B**=FOUR(**A**)	**A** contains N elements. If **B** is COMPLEX SHORT: $16N$ bytes. If **B** is COMPLEX type, no extra memory is needed.
FNROOT	112.5 bytes if FNROOT is not nested. 96.5 additional bytes for each level of nesting.
INTEGRAL	208.5 bytes if INTEGRAL is not nested. 192.5 additional bytes for each level of nesting.

Appendix C

Error Conditions

The Math Pac reports two classes of error messages.

- Math Pac error messages. These have a LEX ID number of 2. These error messages are explained in the first table.
- HP-71 error messages that are reported by Math Pac functions. These have a LEX ID number of 0. These error messages are explained in the second table.

Math Pac Error Messages

Number	Error Message and Condition
1	#DIMS • DOT(**A**,**B**): **A** or **B** is a matrix. • DET(**A**), MAT **B**=INV(**A**), MAT **B**=TRN(**A**), MAT **A**=IDN, MAT **X**=SYS(**A**,**Y**): **A** or **B** is a vector. • MAT**A**=IDN(*i*): only one redimensioning subscript specified. • MAT**A**=*operation*(*operand array(s)*): number of subscripts of **A** is not the same as the number of subscripts required for the result of the operation.
2	Not Square • DET(**A**), MAT **A**=IDN, MAT **B**=INV(**A**), MAT **X**=SYS(**A**,**B**): **A** is a matrix but the number of rows of **A** is not equal to the number of columns. • MAT **A**=IDN(*i*,*j*): $i \neq j$.

Number	Error Message and Condition
3	Conformability
	• MAT **A**=**B**+**C**, MAT **A**=**B**−**C**: **B** and **C** are not conformable for addition (the number of rows are unequal or the number of columns are unequal)
	• MAT **A**=**B***C: **B** and **C** are not conformable for multiplication (**B** is a vector or the number of columns of **B** is not equal to the number of rows of **C**).
	• MAT **A**=TRN(**B**)*C: **B** and **C** are not conformable for transpose multiplication (**B** is a vector or the number of rows of **B** is not equal to the number of rows of **C**).
	• MAT **X**=SYS(**A**,**B**): Although **A** is a square matrix, **A** and **B** are not conformable for multiplication.
	• DOT(**A**,**B**): Although **A** and **B** are vectors, the number of elements of **A** is not equal to the number of elements of **B**.
4	Parameter Redim
	• The result array of a MAT statement is a subprogram parameter. The statement requires array redimensioning, which changes the number of array elements.
5	Nesting Error
	• More than five FNROOT or INTEGRAL keywords are nested.
6	Kybd FN in FNROOT/INTEGRAL
	• Attempting to execute FNROOT or INTEGRAL from the keyboard in BASIC mode, and the function whose root or integral is sought is a user-defined function.
	• Attempting to execute a user-defined function from the keyboard while an FNROOT or INTEGRAL execution is suspended during the evaluation of the function whose root or integral is sought.
7	Function Interrupted
	• Interrupting DET(**A**), CNORM(**A**), RNORM(**A**), FNORM(**A**), or DOT(**A**,**B**) by pressing ATTN twice.
8	Bad Array Size
	• MAT **B**=FOUR(**A**) where the number of elements of **A** is not a non-negative integral power of two.
	• MAT **B**=PROOT(**A**) where **A** has only one element.
9	PROOT Failure
	• PROOT failed to find a root.

Number	Error Message and Condition
10	`GAMMA=Inf` • `GAMMA(X)` where *X* is a non-positive integer.
11	`ATANH(+-1)` • `ATANH(1)` or `ATANH(-1)`
No error number	`Initialization` ■ The Math ROM cannot initialize due to insufficient memory. This ROM requires 43.5 bytes of user memory for its own use. This memory must be available before plugging in the module.

HP-71 Error Messages

Number	Error Message and Condition
11	`Invalid Arg` ■ `BVAL(B$,R)`, `BSTR$(X,R)`: The rounded integer value of *R* is not equal to 2, 8, or 16. • `BVAL(B$,R)`: *B$* is not a valid string representation of a number in base *R*. ■ `BSTR$(X,R)`: The rounded integer value of *X* is not in the interval [0,1E12). ■ `BVAL(B$,R)`: The decimal equivalent of *B$* exceeds 999,999,999,999. ■ `LBND(A,N)`, `UBND(A,N)`: The rounded integer value of *N* is not equal to 1 or 2. ■ An illegal subscript has been used in a `MAT CON`, `MAT IDN`, `MAT ZER`, `COMPLEX`, or `COMPLEX SHORT` statement.
24	`Insufficient Memory` ■ Appendix B gives the memory requirements for various Math Pac operations.
31	`Data Type` ■ A scalar (real or complex) has been used where an array is required or vice-versa. • A complex type (scalar or array) has been used where a real type (scalar or array) is required or vice-versa.

Number	Error Message and Condition
32	No Data
	• Attempting to execute DETL before the first completion of MAT . . . INV with a real-type argument or MAT . . . SYS with a real-type first argument.
	• Attempting to execute FVALUE or FGUESS before the first completion of an FNROOT keyword.
	• Attempting to execute IVALUE or IBOUND before any INTEGRAL keyword has completed the first evaluation of the function whose integral is sought.
	• Attempting to execute FVAR while no FNROOT is evaluating the function whose root is sought.
	• Attempting to execute IVAR while no INTEGRAL is evaluating the function whose integral is sought.
46	Invalid USING
	• Formatting a real expression with a complex IMAGE field or vice-versa.
79	Illegal Context
	• Attempting to execute INTEGRAL or FNROOT from CALC mode in any way except by direct execution.
80	Invalid Parameter
	• MAT INPUT attempts to execute an expression in the MAT INPUT response line where that expression calls a user-defined function.

Appendix D

Attention Key Actions

The way [ATTN] operates during the execution of each the following three keywords is described on the referenced page.

MAT INPUT	Refer to page 54.
FNROOT	Refer to page 97.
INTEGRAL	Refer to page 111.

The keywords listed below in this appendix can be aborted by pressing the [ATTN] key once or twice.

Array Output Statements

All Math Pac array output statements (MAT DISP/PRINT[USING]) can be halted at any time by pressing [ATTN] *once*.

Other MAT Statements

The following MAT statements may be halted at any time by pressing [ATTN] *twice*.

MAT *result* = [−] *operand*

MAT *result* = *operand* +/−/* *operand*

MAT *result* = (*scalar*) [* *operand*]

MAT *result* = INV(*operand*)

MAT *result* = SYS(*operand* , *operand*)

MAT *result* = TRN(*operand*)[* *operand*]

MAT *result* = FOUR(*operand*)

MAT *result* = PROOT(*operand*)

Suppose a lengthy program contained a MAT INV statement. Suppose further that you wished to abort this program. You press [ATTN] once, and the program does not halt (the **SUSP** annunciator does not turn on). This tells you that the MAT INV statement may be executing, and gives you a chance to wait for the result of this MAT INV execution, or to abort the MAT INV execution and the program immediately by pressing [ATTN] a second time. In this way the "press [ATTN] twice" rule gives a user more control over program and statement suspension.

Pressing [ATTN] once during execution of MAT INV would suspend the program in the usual way after this statement is completed.

Scalar-Valued Array Functions

The following scalar-valued array functions can be halted at any time by pressing [ATTN] *twice.*

DET(*operand*)

DOT(*operand* , *operand*)

FNORM(*operand*)

CNORM(*operand*)

RNORM(*operand*)

The benefits provided by this "press [ATTN] twice" rule are the same as those described above. However, only an error can halt the execution of an expression, so when you press [ATTN] twice to halt any of the above functions, the HP-71 will display the error message Function Interrupted.

Appendix E

Numeric Exceptions and the IEEE Proposal

Introduction

This appendix will discuss IEEE exception handling by Math Pac functions and operations, including computation with NaN and Inf arguments, exception flag setting, handling of out-of-range arguments, error or warning messages, and default values for IVL and DVZ exceptions. The HP-71 reference manual discusses the IEEE proposal for handling math exceptions. Math Pac functions, when appropriate, will set the exception flags IVL, DVZ, OVF, UNF, and INX and report errors or warnings (with default results returned) according to the TRAP settings for each of these flags. You can refer to the appropriate sections of this manual for definitions and/or computational formulas for many of the functions described here.

No exception flags are set by any of the keywords in sections 2 or 3 of this manual, or by Math Pac keywords TYPE, - (negation of complex numbers), CONJ, CON, IDN, ZER, MAT DISP/PRINT[USING], LBND, UBND, DETL, FVAR, FVALUE, FGUESS, IVAR, IVALUE, and IBOUND. Remember that exception flags INX, OVF, and UNF may be set when values are rounded to fit the destination type, such as, for example, assigning (MAXREAL,MAXREAL) to a COMPLEX SHORT variable or executing MAT **A**=**B** where **A** is INTEGER type and **B** contains elements greater than 99999.

Aside from exceptions occurring during rounding, the statements MAT **A**=**B**, MAT **A**=-**B**, MAT **A**=TRN(**B**), and MAT **A**=(*X*) set only the IVL exception flag (reporting message Signaled Op) and only when **A** is INTEGER type and either **B** contains, or *X* is, a signaling NaN. This is because INTEGER variables can contain only quiet, not signaling, NaNs. The same applies to MAT INPUT.

The cases given for each of the keywords in the tables which follow are evaluated in order from top to bottom.

Note: Throughout this appendix, ⋆ represents any argument.

Scalar Functions

These functions are described in section 4 of this manual. Any signaling NaN argument sets IVL and reports message Signaled Op; if TRAP(IVL) = 2, then this NaN becomes quiet and the operation can continue. With the exception of the NaN# function, any quiet NaN argument returns a NaN result with no exception flags set. (Aside from signaling NaN arguments, the functions IROUND and NaN# set no exception flags).

Real Hyperbolic Sine (SINH(*X*))

Argument *X*	Result
±Inf	*X*; no exception flags set.
±0	*X*; no exception flags set.
☆	INX set; UNF, OVF set as appropriate.

Real Hyperbolic Cosine (COSH(*X*))

Argument *X*	Result
±Inf	|*X*|; no exception flags set.
±0	1; no exception flags set.
☆	INX set; OVF set as appropriate.

Real Hyperbolic Tangent (TANH(*X*))

Argument *X*	Result
±Inf	SGN(*X*); no exception flags set.
±0	*X*
☆	INX set; UNF set as appropriate.

Real Hyperbolic Arc Sine (ASINH(*X*))

Argument *X*	Result
±Inf	*X*; no exception flags set.
±0	*X*; no exception flags set.
☆	INX set; UNF set as appropriate.

Real Hyperbolic Arc Cosine (ACOSH(*X*))

Argument *X*	Result
Inf	*X*; no exception flags set.
$X < 1$	IVL set; NaN result; message Invalid Arg.
1	0; no exception flags set.
⋆	INX set.

Real Hyperbolic Arc Tangent (ATANH(*X*))

Argument *X*	Result
$\|X\| > 1$	IVL set; NaN result; message Invalid Arg.
$\|X\| = 1$	DVZ set; message ATANH(+-1). SGN(*X*) × Inf result if TRAP(DVZ) = 2. SGN(*X*) × MAXREAL result with INX set if TRAP(DVZ) = 1.
±0	*X*; no exception flags set.
⋆	INX set; UNF set as appropriate.

Base 2 Logarithm (LOG2(*X*))

Argument *X*	Result
Inf	*X*; no exception flags set.
$X < 0$	IVL set; NaN result; message LOG(neg).
±0	DVZ set; message LOG(0). −Inf result if TRAP(DVZ) = 2. −MAXREAL result with INX set if TRAP(DVZ) = 1.
1	0; no exception flags set.
⋆	INX set.

Gamma Function (GAMMA(*X*))

Argument *X*	Result
Inf	*X*; no exception flags set.
±0	DVZ set; message GAMMA=INF. CLASS(*X*) × Inf result if TRAP(DVZ) = 2. CLASS(*X*) × MAXREAL result with INX set if TRAP(DVZ) = 1.
$X < 0$ and integral	DVZ set; message GAMMA=INF. −Inf result if TRAP(DVZ) = 2. −MAXREAL result with INX set if TRAP(DVZ) = 1.
⋆	INX set for all *X* not in the set {1, 2, ..., 18}; UNF, OVF set as appropriate.

Nearest Machine Number (NEIGHBOR(*X*, *Y*))

Arguments		Result
X	*Y*	
X = *Y*	*X* = *Y*	*X*; UNF, INX set if TRAP(UNF) ≠ 2 and 0 < |*X*| < EPS.
MAXREAL	Inf	*Y*; no exception flags set.
−MAXREAL	−Inf	*Y*; no exception flags set.
±Inf	✱	SGN(*X*) × MAXREAL; no exception flags set.
±0	✱	SGN(*Y*) × MINREAL; UNF, INX set if TRAP(UNF) ≠ 2.
MINREAL	±0	0; no exception flags set.
−MINREAL	±0	−0; no exception flags set.
✱	✱	UNF, INX set if |NEIGHBOR(X, Y)| < EPS and TRAP(UNF) ≠ 2.

Power of Ten Scaling (SCALE10(*X*, *N*))

Arguments		Result
X	*N*	
✱	non-integer	IVL set; NaN result; message Invalid Arg.
±Inf	−Inf	IVL set; NaN result; message Inf✱0.
0	Inf	IVL set; NaN result; message Inf✱0.
±Inf	✱	*X*; no exception flags set.
✱	−Inf	SGN(*X*) × 0; no exception flags set.
✱	Inf	SGN(*X*) × Inf; no exception flags set.
✱	✱	INX, OVF, UNF set as appropriate.

Complex Functions and Operations

These functions are described in section 5 of this manual. For extensions of HP-71 and Math Pac functions to complex arguments (+, −, ✱, /, ^, LOG, EXP, SIN, COS, TAN, SINH, COSH, TANH, SQRT, SGN, ABS, =, <, >, ?, and #), only the complex case is discussed here. For the functions POLAR, RECT, ARG, and PROJ, computation at a real argument X is equivalent to computation at the complex argument $(X, 0)$.

Any signaling NaN argument (including real and imaginary parts of complex arguments) sets IVL and reports message Signaled Op; if TRAP(IVL) = 2, then this NaN becomes quiet and the operation can continue. In the following discussion, all references to NaNs are to quiet NaNs.

The following terms are used:

- *Complex* denotes complex DATA type.
- *Real* denotes real DATA type (e.g., (3, 0) is complex and 3 is real).
- CNaN denotes any complex number with at least one NaN component.
- CInf denotes any complex number whose magnitude is Inf; that is, any complex number with at least one ±Inf component.
- *CZERO* denotes any complex number whose magnitude is 0.
- *Arg(Z)* denotes the argument of Z, that is, the infinitely precise value of the Math Pac function ARG(Z).
- $|Z|$ denotes the magnitude of Z.
- The complex variables Z and W will also be denoted by (x, y) and (u, v) respectively.

+, − (Addition and Subtraction)

For real a and complex Z, $a \pm Z = (a \pm x, y)$ and $Z \pm a = (x \pm a, y)$. For complex Z and W, $Z \pm W = (x \pm u, y \pm v)$. IVL is set and message Inf-Inf is reported if any componentwise addition or subtraction is equivalent to Inf − Inf; a NaN is returned for the corresponding result component. Otherwise, INX, OVF, and UNF are set for each result component as appropriate.

* (Multiplication)

For real a and complex Z, $a \times Z = Z \times a = (ax, ay)$. IVL is set and message Inf*0 is reported if any componentwise multiplication is equivalent to $(\pm \text{Inf}) \times (\pm 0)$; a NaN is returned for the corresponding result component. Otherwise, INX, OVF, and UNF are set for each result component as appropriate.

For complex Z and W, $Z \times W$ is given by the table below.

Complex × Complex Multiplication ($Z * W$)

Arguments		Result
Z	*W*	
CNaN	⋆	(NaN,NaN); no exception flags set.
⋆	CNaN	(NaN,NaN); no exception flags set.
CInf	CZERO	IVL set; (NaN,NaN) result; message Inf*0.
CZERO	CInf	IVL set; (NaN,NaN) result; message Inf*0.
CInf	⋆	RECT((Inf, Arg(*Z*) + Arg(*W*))); no exception flags set.
⋆	CInf	RECT((Inf, Arg(*Z*) + Arg(*W*))); no exception flags set.
⋆	⋆	($xu - yv$, $xv + yu$); INX, OVF, UNF set for each result component as appropriate.

/ (Division)

For real a and complex Z, $Z/a = (x/a, y/a)$. IVL is set and message 0/0 is reported if any componentwise division is equivalent to $(\pm 0)/(\pm 0)$; a NaN is returned for the corresponding result component. IVL is set and message Inf/Inf is reported if any componentwise division is equivalent to $(\pm \text{Inf})/(\pm \text{Inf})$; a NaN is returned for the corresponding result component. DVZ is set and message /Zero is reported if any componentwise division is equivalent to $T/(\pm 0)$ where T is neither a NaN, $\pm$Inf, or ± 0; Inf of the appropriate sign is returned for the corresponding result component if TRAP(DVZ) = 2; MAXREAL of the appropriate sign is returned with INX set for the corresponding result component if TRAP(DVZ) = 1. Otherwise, INX, OVF, and UNF are set for each result component as appropriate.

For complex Z, we define the following. If Z = CZERO, then $1/Z$ is defined to be (CLASS(x) × Inf, −SGN(y)). If Z = CInf, then $1/Z$ is defined to be (SGN(x) × 0, −SGN(y) × 0).

For real a and complex Z, a/Z is given by the table below.

Real/Complex Division (*a* / *Z*)

Arguments		Result
a	*Z*	
NaN	⋆	⟨NaN,NaN⟩; no exception flags set.
⋆	CNaN	⟨NaN,NaN⟩; no exception flags set.
±Inf	CInf	IVL set; ⟨NaN,NaN⟩ result; message Inf/Inf
±0	CZERO	IVL set; ⟨NaN,NaN⟩ result; message 0/0.
±Inf	CZERO	SGN(*a*) × (1/*Z*) (real × complex multiplication); no exception flags set.
⋆	CZERO	DVZ set; message /Zero. *a* × (1/*Z*) (real × complex multiplication) result if TRAP⟨DVZ⟩ = 2. *a* × (1/*Z*) (real × complex multiplication) result with ±Inf result component replaced by ±MAXREAL and INX set if TRAP⟨DVZ⟩ = 1.
⋆	CInf	■ × (1/*Z*) (real × complex multiplication); no exception flags set.
±Inf	⋆	*a* × CONJ⟨*Z*⟩ (real × complex multiplication); IVL set and message Inf*0 reported if any componentwise multiplication is equivalent to (±Inf) × (±0); a NaN is returned for the corresponding result component. Otherwise, no exception flags set.
⋆	⋆	($a/|Z|^2$) × CONJ⟨*Z*⟩ (real × complex multiplication); INX, OVF, UNF set for each result component as appropriate.

For complex Z and W, W/Z is given by the table below.

Complex / Complex Division (W/Z)

Arguments		Result
W	*Z*	
CNaN	⋆	⟨NaN, NaN⟩; no exception flags set.
⋆	CNaN	⟨NaN, NaN⟩; no exception flags set.
CZERO	CZERO	IVL set; ⟨NaN, NaN⟩ result; message 0/0.
CInf	CInf	IVL set; ⟨NaN, NaN⟩ result; message Inf/Inf.
CInf	CZERO	$W \times (1/Z)$ (complex × complex multiplication); no exception flags set.
⋆	CZERO	DVZ set; message /Zero. $W \times (1/Z)$ (complex × complex multiplication) result if TRAP⟨DVZ⟩ = 2. $W \times (1/Z)$ (complex × complex multiplication) result with ±Inf result component(s) replaced by ±MAXREAL and INX set if TRAP⟨DVZ⟩ = 1.
⋆	CInf	$W \times (1/Z)$ (complex × complex multiplication); no exception flags set.
⋆	⋆	$(W \times \text{CONJ}(Z))/\lvert Z\rvert^2$ (complex × complex multiplication and complex/real division); INX, OVF, UNF set for each result component as appropriate.

For complex Z, $f(Z)$ is given for the specified functions by the following tables.

Complex Sine (SIN⟨Z⟩)

Argument *Z*	Result
CNaN	⟨NaN, NaN⟩; no exception flags set.
(±Inf, ⋆)	IVL set; ⟨NaN, NaN⟩ result; message Invalid Arg.
(⋆, ±Inf)	RECT⟨⟨Inf, Arg((sin(x), SGN⟨y⟩cos(x)))⟩⟩; no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Complex Hyperbolic Sine (SINH⟨Z⟩)

Argument *Z*	Result
CNaN	⟨NaN, NaN⟩; no exception flags set.
(⋆, ±Inf)	IVL set; ⟨NaN, NaN⟩ result; message Invalid Arg.
(±Inf, ⋆)	RECT⟨⟨Inf, Arg((SGN⟨x⟩cos(y), sin(y)))⟩⟩; no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Complex Cosine (COS(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
(±Inf, ⋆)	IVL set; (NaN,NaN) result; message Invalid Arg.
(⋆, ±Inf)	RECT((Inf, Arg((cos(*x*), -SGN(*y*)sin(*x*))))); no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Complex Hyperbolic Cosine (COSH(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
(⋆, ±Inf)	IVL set; (NaN,NaN) result; message Invalid Arg.
(±Inf, ⋆)	RECT((Inf, Arg((cos(*y*), SGN(*x*)sin(*y*))))); no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Complex Tangent (TAN(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
(±Inf, ±Inf)	(0, SGN(*y*)); no exception flags set.
(±Inf, ⋆)	IVL set; (NaN,NaN) result; message Invalid Arg.
(⋆, ±Inf)	(SGN(sin(*x*)cos(*x*))*0, SGN(*y*)); no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Complex Hyperbolic Tangent (TANH(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
(±Inf, ±Inf)	(SGN(*x*),-0); no exception flags set.
(⋆, ±Inf)	IVL set; (NaN,NaN) result; message Invalid Arg.
(±Inf, ⋆)	(SGN(*x*),SGN(sin(*y*)cos(*y*))*0); no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Absolute Value (ABS(*Z*))

Argument *Z*	Result
CNaN	NaN; no exception flags set.
CInf	Inf; no exception flags set.
⋆	INX, OVF, UNF set as appropriate.

Argument (ARG(*Z*))

Argument *Z*	Result
CNaN	NaN; no exception flags set.
(Inf, Inf)	45 degrees or $\pi/4$ radians; INX set if radian mode.
(−Inf, Inf)	135 degrees or $3\pi/4$ radians; INX set if radian mode.
(Inf, −Inf)	−45 degrees or $-\pi/4$ radians; INX set if radian mode.
(−Inf, −Inf)	−135 degrees or $-3\pi/4$ radians; INX set if radian mode.
⋆	ANGLE(*x*, *y*); INX, UNF set as appropriate.

Projective Infinity (PROJ(*Z*))

Argument *Z*	Result
CNaN	(NaN, NaN); no exception flags set.
CInf	(Inf, 0); no exception flags set.
⋆	*Z*; UNF, INX set for any component whose magnitude is between 0 and EPS if TRAP(UNF) ≠ 2.

Unit Vector (SGN(*Z*))

Argument *Z*	Result
CNaN	(NaN, NaN); no exception flags set.
CZERO	*Z*; no exception flags set.
(±Inf, ±Inf)	RECT((1, Arg(*Z*))); INX set.
(±Inf, ⋆)	(SGN(*x*), SGN(*y*)*0); no exception flags set.
(⋆, ±Inf)	(SGN(*x*)*0, SGN(*y*)); no exception flags set.
⋆	INX, UNF set for each result component as appropriate.

Square Root (SQRT(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
CInf	RECT((Inf, Arg(Z)/2)); no exception flags set.
⋆	INX, UNF set for each result component as appropriate.

Rectangular to Polar Conversion (POLAR(Z))

Argument Z	Result
⋆	(ABS(Z),ARG(Z)); INX, OVF, UNF set for each result component as appropriate.

Polar to Rectangular Conversion (RECT(Z))

Argument Z	Result
CNaN	(NaN,NaN); no exception flags set.
(±Inf, ±Inf)	(SGN(x)*Inf,0); no exception flags set.
(±0, ±Inf)	(x, x); no exception flags set.
(⋆, ±Inf)	IVL set; (NaN,NaN) result; message Invalid Arg.
(±Inf, ⋆)	(acos(y),bsin(y)); no exception flags set; $a = \begin{cases} x \text{ if } \cos(y) \neq 0 \\ \text{SGN}(x) \text{ if } \cos(y) = 0 \end{cases}$ and $b = \begin{cases} x \text{ if } \sin(y) \neq 0 \\ \text{SGN}(x) \text{ if } \sin(y) = 0 \end{cases}$
⋆	(xcos(y),xsin(y)); INX, UNF set for each result component as appropriate.

Natural Logarithm (LOG(*Z*))

Argument *Z*	Result
CNaN	(NaN,NaN); no exception flags set.
CZERO	DVZ set; message LOG(0). (-Inf, ARG(*Z*)) result if TRAP(DVZ) = 2. (-MAXREAL,ARG(*Z*)) result with INX set if TRAP(DVZ) = 1.
CInf	(Inf,ARG(*Z*)); INX set for the result imaginary part as appropriate.
⋆	INX, UNF set for each result component as appropriate.

Exponential (EXP(*Z*))

Argument *Z*	Result
CNaN	(NaN,NaN); no exception flags set.
(-Inf, ±Inf)	(0,0); no exception flags set.
(Inf, ±Inf)	(Inf,0); no exception flags set.
(⋆, ±Inf)	IVL set; (NaN,NaN) result; message Invalid Arg.
(-Inf, ⋆)	(0 × cos(*y*),0 × sin(*y*)); INX set for each result component ■ appropriate.
(Inf, ⋆)	RECT(*Z*); no exception flags set.
⋆	INX, OVF, UNF set for each result component as appropriate.

Relational Operators

When comparing two values, at least one of which is complex, any numeric comparison operator containing < or > without ? or # sets IVL and reports message Unordered. If TRAP(IVL) = 2, then a result of 0 or 1 will be returned based on the presence of the comparison operator =, that is, $Z <= W$, $Z >= W$, and $Z <>= W$ are true if and only if $x = u$ and $y = v$; $Z < W$, $Z > W$, and $Z <> W$ are always false.

^ (Exponentiation)

Before $W \hat{} Z$ is computed, the following peliminary actions are taken:

1. If either W or Z is real then, for the purposes of the computation, it becomes complex with 0 imaginary part.
2. If either W or Z is a CNaN, then a result of (NaN,NaN) is returned with no exception flags set.
3. For the purposes of the computation, W and Z are then converted to a canonical form representation defined as follows: if one part of a complex number is ±Inf while the other part is finite, then the canonical form representation replaces the finite part by ±0 (that is, preserves its sign); otherwise, the complex number is already said to be in canonical form. For example, (0,Inf) and (-Inf,-0) are the canonical form representations of (6.7,Inf) and (-Inf,-MAXREAL) respectively. In what follows, W and Z are assumed to be in canonical form.

For W = CZERO, $W \hat{} Z$ is given by the table below.

Exponention ($W \hat{} Z$): W = CZERO

Argument Z	Result
$x > 0$	(SGN(*u*^*x*),0); no exception flags set.
$x < 0$	DVZ set; message 0^Neg. (SGN(*u*^*x*)*Inf,0) result if TRAP(DVZ) = 2. (SGN(*u*^*x*)*MAXREAL,0) result with INX set if TRAP(DVZ) = 1.
$x = 0$ and $y = 0$	No exception flags set; message 0^0 reported; default result of (1,0) returned if TRAP(IVL) ≠ 0.
$x = 0$ and $y \neq 0$	IVL set; (NaN,NaN) result; message Invalid Arg.

For $y \neq 0$, $W \hat{} Z$ is given by the table below.

Exponention ($W \hat{} Z$): $y \neq 0$

Arguments		Result
W	Z	
(1, ±0)	CInf	IVL set; (NaN,NaN) result; message 1^Inf.
⋆	⋆	EXP(*Z*LOG(*W*)) (complex × complex multiplication). If *Z*LOG(*W*) equals (±0, ±Inf), then this quantity is not in the domain of EXP and IVL is set, (NaN,NaN) is returned, and message Invalid Arg is reported. Otherwise, INX, OVF, and UNF are set for each result component as appropriate.

For $y = 0$ and $v \neq 0$, $W \wedge Z$ is given by the table below.

Exponention ($W \wedge Z$): $y = 0$ and $v \neq 0$

Arguments		Result
W	*Z*	
$\|W\| = 1$	CInf	IVL set; (NaN,NaN) result; message Invalid Arg.
CInf	CZERO	No exception flags set; message Inf^0 reported; default result of (1,0) returned if TRAP(IVL) ≠ 0.
☆	☆	EXP(x*LOG(W)) (real • complex multiplication); INX, OVF, UNF set for each result component as appropriate.

For $y = 0$ and $v = 0$, $W \wedge Z$ is given by the table below.

Exponention ($W \wedge Z$): $y = 0$ and $v = 0$

Arguments		Result
W	*Z*	
$u = \pm$Inf	$x = 0$	No exception flags set; message Inf^0 reported; default result of (1,0) returned if TRAP(IVL) ≠ 0.
$u = \pm 1$	CInf	IVL set; (NaN,NaN) result; message 1^Inf.
☆	CInf	(|u|^x,0); no exception flags set.
☆	☆	EXP(x*LOG(W)) (real × complex multiplication); INX, OVF, UNF set for each result component as appropriate.

Array Functions and Operations

These functions are described in sections 7, 8, and 9 of this manual. Refer to the previous discussion for definitions of CZERO, CInf, complex, etc.

CNORM(**A**), RNORM(**A**)

If **A** is $M \times N$ (for vectors take $N = 1$), then

$$\text{CNORM}(\mathbf{A}) = \underset{1<=j<=N}{\text{MAX}} \sum_{i=1}^{M} |a_{ij}| \qquad \text{RNORM}(\mathbf{A}) = \underset{1<=j<=M}{\text{MAX}} \sum_{j=1}^{N} |a_{ij}|$$

If any element of **A** is a signaling NaN (including either part of complex array elements), then each function sets IVL and reports message Signaled Op. If TRAP(IVL) = 2, the result is a quiet NaN with no other elements processed.

If any element of **A** is a quiet NaN (including either part of complex array elements), then each function sets IVL and reports message Unordered; a NaN result is returned. Otherwise, INX, OVF, and UNF are set for the result as appropriate.

FNORM(**A**)

If **A** is $M \times N$ (for vectors take $N = 1$), then

$$\text{FNORM}(\mathbf{A}) = \sqrt{\left(\sum_{i=1}^{M} \sum_{j=1}^{N} |a_{ij}|^2\right)}$$

If any element of **A** is a signaling NaN (including either part of complex array elements), then IVL is set and message Signaled Op is reported. If TRAP(IVL) = 2, the result is a quiet NaN with no other elements processed.

Quiet NaNs pass through with no exception flags set. Otherwise, INX, OVF, and UNF are set for the result as appropriate.

DOT(A,B)

If **A** and **B** are N-element vectors, then

$$\texttt{DOT(A,B)} = \sum_{i=1}^{N} \overline{a_i} b_i$$

(If either **A** or **B** is complex, refer to the definitions of complex addition and multiplication given previously). If any element of **A** or **B** is a signaling NaN (including either part of complex array elements), then IVL is set with message Signaled Op. If, in any term in the above expression, ± 0 or CZERO is multiplied by $\pm$Inf or CInf, then IVL is set with message Inf*0. If, in the above expression, the summation executes an addition equivalent to Inf − Inf, then IVL is set with message Inf-Inf.

If only one IVL exception occurs, that message is reported. If more than one IVL exception occurs, the particular message(s) reported depends upon the order and type of exception that occurs. If TRAP(IVL) = 2, the result is either a real NaN or a complex value with one or two NaN components. Quiet NaNs pass through with no exception flags set. Otherwise, INX, OVF, and UNF are set for the result, or each result component, as appropriate.

MAT C=A*B

If **A** is $M \times N$ and **B** is $N \times P$ (for vectors take $P=1$), then

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

(If either **A** or **B** is complex, refer to the definitions of complex addition and multiplication given previously). Since each result element is derived from an inner product, exception handling is the same as that for DOT(**A**,**B**), applied to each result element separately.

MAT C=TRN(A)*B

If **A** is $M \times N$ and **B** is $M \times P$ (for vectors take $P=1$), then

$$c_{ij} = \sum_{k=1}^{M} \overline{a_{ki}} b_{kj}$$

(If either **A** or **B** is complex, refer to the definitions of complex addition and multiplication given previously).

Since each result element is derived from an inner product, exception handling is the same as that for DOT(**A**,**B**), applied to each result element separately.

MAT C=A±B

All elements of **C** are computed separately as

$$c_{ij} = a_{ij} \pm b_{ij}$$

(If either **A** or **B** is complex, refer to the definitions of complex addition and subtraction given previously).

If any element of **A** or **B** is a signaling NaN (including either part of complex array elements), then IVL is set and message Signaled Op is reported. If TRAP(IVL) = 2, the corresponding result element or component becomes a quiet NaN and the operation continues. Quiet NaNs pass through with no exception flags set.

IVL is set and message Inf-Inf is reported if any addition or subtraction (or componentwise addition or subtraction) is equivalent to Inf − Inf; a NaN is returned for the corresponding result element or component. Otherwise, INX, OVF, and UNF are set for each result element or component as appropriate.

MAT B=(*s*)*A

All elements of **B** are computed separately as

$$b_{ij} = sa_{ij}$$

(If either s or **A** is complex, refer to the definition of complex multiplication given previously). If s (or either part of s, if s is complex) is a signaling NaN, then IVL is set and message Signaled Op is reported; if any element of **A** is a signaling NaN (including either part of complex array elements), then IVL is set and message Signaled Op is reported. In either event, if TRAP(IVL) = 2, these NaNs become quiet and the operation continues. Quiet NaNs pass through with no exception flags set.

IVL is set and message Inf*0 is reported if, during the computation of any result element, ±0 or CZERO is multiplied by ±Inf or CInf. If TRAP(IVL) = 2, the corresponding result element is either a real NaN or a complex value with one or two NaN components. Otherwise, INX, OVF, and UNF are set for each result element or component as appropriate.

DET(A), MAT C=INV(A), MAT C=SYS(A,B)

Due to the intricate algorithmic basis of these three operations, exception handling is complex; only a summary is provided here.

If any element of **A** or **B** is a signaling NaN (including either part of complex array elements), then IVL is set and message Signaled Op is reported. If TRAP(IVL) = 2, the corresponding element or component becomes a quiet NaN and the operation continues. Quiet NaNs pass through with no exception flags set.

OVF, UNF, and INX are set for each result element as appropriate and may also be set at intermediate stages of the computation (especially OVF when **A** is (machine) singular). IVL may also be set with any of the following messages reported: Inf*0, Inf-Inf, and/or Inf/Inf. These messages are only possible due to a ±Inf in **A** or **B** or an intermediate overflow becoming ±Inf; in the latter case they may be suppressed by setting TRAP(OVF) = 1 before the computation.

Other Math Pac Functions

PROOT

Special cases for the PROOT function are handled first. These are NaNs, Infs, or leading and trailing zeros in the coefficient array.

NaNs are handled first. If any coefficient is a NaN, then every element of the result array becomes (NaN,NaN) with no exception flags set and the function is complete. (Signaling NaN coefficients do not set IVL).

Infs are dealt with next. If any coefficient is ±Inf, then every finite coefficient will become zero and the computation falls through to handle leading and trailing zeros.

Leading zeros are handled next. Every leading zero coefficient will produce a root at (Inf,Inf) with no exception flags set. The next coefficient then becomes the leading coefficient and the process loops. Every such root stored decrements the degree of the polynomial; the function is complete if the degree becomes zero.

Trailing zeros are handled next. Every trailing zero coefficient will produce a root at (0,0) with no exception flags set. The second to the last coefficient then becomes the trailing coefficient and the process loops. Every such root stored decrements the degree of the polynomial and the function is complete if the degree becomes zero.

At this point, the degree of the polynomial is positive and either all (remaining) coefficients are finite, in which case the roots of the (reduced) polynimial will be found, or the leading and trailing coefficients are both ±Inf. In the latter case, at least two of the original coefficients were ±Inf and factorization does not make sense; if the (new) degree of the polynomial is D, then D roots at (NaN,NaN) are stored into the result array and the function is complete; every such root stored sets IVL and reports message Invalid Arg.

Except for the above special cases, OVF and UNF are set for every result array component as appropriate with INX always set.

FOUR

As with the PROOT function, special cases for the FOUR function are handled first. These are NaN and Inf components in the data array.

NaNs are handled first. If any component of any data array element is a NaN, then every element of the result array becomes (NaN,NaN) with no exception flags set and the function is complete. (Signaling NaN components do not set IVL).

Infs are dealt with next. If any component of any data array element is ±Inf, then every result element becomes (Inf,Inf) with no exception flags set and the function is complete.

Except for the above special cases, OVF and UNF are set for every result array component as appropriate with INX always set unless the data array was identically zero.

FNROOT and INTEGRAL

If a NaN (signaling or quiet) results during the evaluation of any of the arguments of FNROOT or INTEGRAL, then error Invalid Arg is reported; no exception flags are set and this error halts the computation.

In general, any value of ±Inf resulting from the evaluation of any of the arguments of FNROOT or INTEGRAL becomes ±MAXREAL for the purposes of the computation. INX, OVF, and UNF are set for the result as appropriate.

Remember that FNROOT looks at the value of TRAP(UNF) to decide whether or not to search the range of denormalized numbers for a root. This region is searched only if TRAP(UNF) = 2 when the FNROOT function is started.

Keyword Index

Keyword	Page	Description
IVALUE	102	Current approximation to an INTEGRAL.
IVAR	102	Variable of integration in INTEGRAL.
LBND	72	Array subscript lower bound.
LBOUND	72	Same as LBND.
LOG	37	Complex natural logarithm.
LOG2	29	Log base 2.
MAT DISP	54	Array display (unformatted).
MAT DISP USING	55	Array display (formatted).
MAT INPUT	53	Interactive array input.
MAT...CON	52	Constant array with redimensioning.
MAT...IDN	52	Identity matrix with redimensioning.
MAT...ZER	53	Zero array with redimensioning.
MAT...ZERO	53	Same as MAT...ZER.
MAT...PRINT	55	Array printing (unformatted).
MAT PRINT USING	56	Array printing (formatted).
MAT =	51	Array copying (simple assignment).
MAT =-	63	Array negation.
MAT =...+	64	Array addition.
MAT =...-	64	Array subtraction.
MAT =...*	65	Array multiplication.
MAT =()	52	Scalar to array assignment (numeric expression assignment).
MAT =()*	65	Scalar multiplication.
MAT = FOUR	135	Finite Fourier Transform.
MAT = INV	77	Matrix inversion.
MAT = PROOT	120	Polynomial rootfinding.
MAT = SYS	79	System solution.
MAT = TRN	77	Transpose or conjugate transpose.
MAT = TRN... *	66	Transpose or conjugate transpose multiply.
NAN$	30	NaN diagnostic function.
NEIGHBOR	30	Successor/predecessor function.
POLAR	40	Rectangular to polar conversion.
PROJ	42	Conversion of complex infinities to projective infinities.
RECT	40	Polar to rectangular conversion.
REPT	21	Real part of complex number.
RNORM	70	Infinity (row) norm of an array.
SCALE10	29	Exponent scaling function.
SGN	41	Complex unit vector.
SIN	38	Complex sine.

Portable Computer Division
1000 N.E. Circle Blvd., Corvallis, OR 97330, U.S.A.

European Headquarters
150, Route Du Nant-D'Avril
P.O. Box, CH-1217 Meyrin 2
Geneva-Switzerland

HP-United Kingdom
(Pinewood)
GB-Nine Mile Ride, Wokingham
Berkshire RG11 3LL

Reorder Number
82480-90001

Printed in Singapore 9/84
82480-90016